


O'REILLY®



ZooKeeper

分布式过程协同技术详解

ZooKeeper

 机械工业出版社
China Machine Press

Flavio Junqueira Benjamin Reed 著
谢超 周贵卿 译

O'Reilly精品图书系列

ZooKeeper: 分布式过程协同技术详解

ZooKeeper: Distributed Process Coordination

[美] 荣凯拉 (Junqueira, F.) [美] 里德 (Reed, B.)

著

谢超 周贵卿 译

ISBN: 978-7-111-52431-1

本书纸版由机械工业出版社于2016年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线: + 86-10-68995265

客服信箱: service@bbbvip.com

官方网址: www.hzmedia.com.cn

新浪微博 @华章数媒

微信公众号 华章电子书（微信号: hzebook）

目录

O'Reilly Media, Inc.介绍

译者序

前言

第一部分 ZooKeeper的概念和基础

第1章 简介

1.1 ZooKeeper的使命

1.2 示例：主-从应用

1.3 分布式协作的难点

1.4 ZooKeeper的成功和注意事项

第2章 了解ZooKeeper

2.1 ZooKeeper基础

2.2 ZooKeeper架构

2.3 开始使用ZooKeeper

2.4 一个主-从模式例子的实现

2.5 小结

第二部分 使用ZooKeeper进行开发

第3章 开始使用ZooKeeper的API

3.1 设置ZooKeeper的CLASSPATH

3.2 建立ZooKeeper会话

3.3 获取管理权

3.4 注册从节点

3.5 任务队列化

3.6 管理客户端

3.7 小结

第4章 处理状态变化

4.1 单次触发器

4.2 如何设置监视点

4.3 普遍模型

4.4 主-从模式的例子

4.5 另一种调用方式: **Multiop**

4.6 通过监视点代替显式缓存管理

4.7 顺序的保障

4.8 监视点的羊群效应和可扩展性

4.9 小结

第5章 故障处理

5.1 可恢复的故障

5.2 不可恢复的故障

5.3 群首选举和外部资源

5.4 小结

第6章 ZooKeeper注意事项

- 6.1 使用ACL
- 6.2 恢复会话
- 6.3 当znode节点重新创建时，重置版本号
- 6.4 sync方法
- 6.5 顺序性保障
- 6.6 数据字段和子节点的限制
- 6.7 嵌入式ZooKeeper服务器
- 6.8 小结

第7章 C语言客户端

- 7.1 配置开发环境
- 7.2 开始会话
- 7.3 引导主节点
- 7.4 行使管理权
- 7.5 任务分配
- 7.6 单线程与多线程客户端
- 7.7 小结

第8章 Curator: ZooKeeper API的高级封装库

- 8.1 Curator客户端程序
- 8.2 流畅式API
- 8.3 监听器
- 8.4 Curator中状态的转换

8.5 两种边界情况

8.6 菜谱

8.7 小结

第三部分 ZooKeeper的管理

第9章 ZooKeeper内部原理

9.1 请求、事务和标识符

9.2 群首选举

9.3 Zab: 状态更新的广播协议

9.4 观察者

9.5 服务器的构成

9.6 本地存储

9.7 服务器与会话

9.8 服务器与监视点

9.9 客户端

9.10 序列化

9.11 小结

第10章 运行ZooKeeper

10.1 配置ZooKeeper服务器

10.2 配置ZooKeeper集群

10.3 重配置

10.4 配额管理

- 10.5 多租赁配置
- 10.6 文件系统布局和格式
- 10.7 四字母命令
- 10.8 通过JMX进行监控
- 10.9 工具
- 10.10 小结

作者介绍

封面介绍

O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

译者序

摩尔定律揭示了集成电路每18个月计算性能就会增加一倍。随着信息的飞速膨胀，很多应用都无法依赖单个服务器的性能升级来处理如此庞大的数据量，分布式系统和应用越来越受到人们的青睐。分布式系统和应用不仅能提供更强的计算能力，还能为我们提供更好的容灾性和扩展性。

在实际开发分布式应用时，开发人员与运维人员都会花费大量时间和精力来处理异构系统中的协作通信问题。这也许并不是你想要的，你最关心的是战略业务是否正常，能否更快更好地提供自己主营业务的系统和服务。因此对分布式系统的协作处理上，需要专门处理协作问题的系统来帮助我们。

ZooKeeper是Google的Chubby项目的开源实现，它曾经作为Hadoop的子项目，在大数据领域得到广泛应用。ZooKeeper以Fast Paxos算法为基础，同时为了解决活锁问题，对Fast Paxos算法进行了优化，因此也可以广泛用于大数据之外的其他分布式系统，为大型分布式系统提供可靠的协作处理功能。比如小米公司的米聊，其后台就采用了ZooKeeper作为分布式服务的统一协作系统。而阿里公司的开发人员也广泛使用ZooKeeper，并对其进行了适当修改，开源了一款TaoKeeper软件，以适应自身业务需要。

本书首先从分布式系统的基本概念入手，然后介绍实际开发编程的接口和技巧，最后谈及运维人员所关心的配置维护知识。翻译过程中，译者对原版书籍通读一遍，对ZooKeeper又有了新的认识和理解，获得了分布式应用构建中需要注意的很多细节，这本书可谓是实际开发和维护中的一本最佳参考书籍。对于这么优秀的一本书，翻译时译者惶恐于译文对读者理解的影响，尽最大努力保持原文意思，以便读者真正能够领悟ZooKeeper的精髓。

由于译者水平有限，译文中的不当之处在所难免，恳请广大读者批评指正。

谢超 周贵卿

前言

构建分布式系统并不容易。然而，人们日常所使用的应用大多基于分布式系统，在短时间内依赖于分布式系统的现状并不会改变。

Apache ZooKeeper旨在减轻构建健壮的分布式系统的任务。**ZooKeeper**基于分布式计算的核心概念而设计，主要目的是给开发人员提供一套容易理解和开发的接口，从而简化分布式系统构建的任务。

即使有了**ZooKeeper**，但开发中分布式处理的环节并不是微不足道的事情，因此我们编写了这本书，通过这本书可以让你快速熟悉如何通过**Apache ZooKeeper**构建分布式系统。我们从基本的概念入手，这样可以使你觉得自己就像是分布式系统的专家一样，在你看到一系列需要注意的警告时，你可能会有一些沮丧，不过不用担心，如果你能够很好地理解我们所阐述的关键点，你已经走在构建良好的分布式系统的正确道路上了。

目标读者

本书适用于分布式系统的开发人员，以及使用**ZooKeeper**进行生产经营的应用程序运维人员。我们假设读者具备**Java**语言的知识，并且本书为读者提供了关于分布式系统中概念的大量背景知识，以便你更好地使用**ZooKeeper**。

本书内容介绍

第一部分阐述了Apache ZooKeeper这类系统的设计目的和动机，并介绍分布式系统的一些必要背景知识。

- 第1章介绍了ZooKeeper可以做什么，以及其设计如何支撑这些任务。

- 第2章介绍了基本概念和基本组成模块，并通过命令行工具的具体操作介绍ZooKeeper可以做什么。

第二部分阐述程序员所需要掌握的ZooKeeper库调用方法和编程技巧，虽然对系统运维人员来说也有一定价值，但也可以不选择阅读。这一部分主要以Java语言的API为主，因为Java是非常流行的开发语言，如果你之前使用其他开发语言，可以通过这一部分内容来学习基本的技术和方法调用，之后通过其他语言来实现。另外，我们也为C语言的应用开发人员提供了一章内容的开发方法。

- 第3章介绍Java语言的API。

- 第4章解释如何跟踪和处理ZooKeeper中的状态变更情况。

- 第5章介绍如何在系统或网络故障时恢复应用。

- 第6章介绍为了避免故障要注意的一些繁杂却很重要的场景。

·第7章介绍C语言版的API，该章也可以作为非Java语言实现的ZooKeeper API的基础，对非Java语言的开发人员非常有帮助。

·第8章介绍一款更高层级的封装的ZooKeeper接口。

第三部分主要适用于ZooKeeper的系统运维人员，尤其在第9章中即便对开发人员也很有价值。

·第9章介绍ZooKeeper的作者们在设计时所采用的方案，这些知识对运维管理非常有帮助。

·第10章介绍如何对ZooKeeper进行配置。

本书约定

本书中采用了以下排版约定：

斜体

用于重点介绍新的术语、URL、命令、工具组件以及文件和目录名称。

等宽字体 (Constant width)

指示变量、方法、类型、参数、对象以及其他代码结构。

等宽加粗 (**Constant width bold**)

指示需要用户输入的命令或其他文本信息，同时也用于命令输出中的重要信息。

等宽斜体 (*Constant width italic*)

指示代码或命令中的占位符，这些占位符需要在实际中替换为合适的值。

注意： 表示一些小窍门、建议或普通注解。

示例代码

代码、练习等附加资料可以到O'Reilly官方网站本书页面下载。

本书用于协助读者构建系统。一般而言，如果本书提供了示例代码，你可以在自己的程序或文档中使用，并不需要联系我们获得授权，除非你复制了大量代码。例如，你在开发程序时使用了本书中的好几处代码则不需要授权，若以CD-ROM方式出售并发布O'Reilly书籍中的示例则需要得到授权许可，引用本书及其示例代码来解答问题并不需要授权许可，将本书中大量示例代码引入你自己的著作中则需要授权许可。

我们非常感谢各类引文参考，但并不强制约束。引文参考包括书名、作者、出版方和ISBN。例如：“ZooKeeper by Flavio Junqueira and Benjamin Reed (O’Reilly) .Copyright 2014Flavio Junqueira and Benjamin Reed, 978-1-449-36130-3.”

如果你对合理使用示例代码时有疑问，或对以上所介绍的许可授权有疑问，请通过permissions@oreilly.com联系我们。

联系我们

有关本书的任何建议和疑问，可以通过下列方式与我们取得联系：

美国：

O'Reilly Media, Inc.

1005Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）

奥莱利技术咨询（北京）有限公司

我们会在本书的网页中列出勘误表、示例和其他信息。可以通过访问如下网址获得：

<http://shop.oreilly.com/product/0636920028901.do>

要评论或询问本书的技术问题，请发送电子邮件到：

bookquestions@oreilly.com

想了解关于O'Reilly图书、课程、会议和新闻的更多信息，请访问以下网站：

<http://www.oreilly.com.cn>

<http://www.oreilly.com>

致谢

我们要感谢本书的编辑人员，从最初的Nathan Jepson到后来Andy Oram的努力，他们的出色工作让我们得以出版此书。

我们要感谢所有在此书上花费这么多时间的家人和雇主，希望你能欣赏我们的成果。

我们要感谢为本书花费大量时间的审阅者们，他们给予我们很大帮助，为我们提供建议来改进本书，他们包括Patrick Hunt、Jordan Zimmerman、Donald Miner、Henry Robinson、Isabel Drost-Fromm和Thawan Kooburat。

ZooKeeper是Apache ZooKeeper社区共同创作的，我们与这些杰出的提交者和贡献者一同工作，非常荣幸能与他们一同工作。我们还要向ZooKeeper的用户们致谢，多年以来，他们向我们提交bug，给予我们很多反馈信息和鼓励。

第一部分 ZooKeeper的概念和基础

这一部分适合任何对ZooKeeper感兴趣的读者，该部分介绍ZooKeeper所处理的问题，以及在ZooKeeper的设计中的权衡取舍。

第1章 简介

在计算机诞生之后很长的一段时间里，一个应用服务是在一个独立的单处理器计算机上运行一段程序。时至今日，应用服务已经发生了很大的变化。在大数据和云计算盛行的今天，应用服务由很多个独立的程序组成，这些独立的程序则运行在形形色色、千变万化的一组计算机上。

相对于开发在一台计算机上运行的单个程序，如何让一个应用中多个独立的程序协同工作是一件非常困难的事情。开发这样的应用，很容易让很多开发人员陷入如何使多个程序协同工作的逻辑中，最后导致没有时间更好地思考和实现他们自己的应用程序逻辑；又或者开发人员对协同逻辑关注不够，只是用很少的时间开发了一个简单脆弱的主协调器，导致不可靠的单一失效点。

ZooKeeper的设计保证了其健壮性，这就使得应用开发人员可以更多关注应用本身的逻辑，而不是协同工作上。**ZooKeeper**从文件系统API得到启发，提供一组简单的API，使得开发人员可以实现通用的协作任务，包括选举主节点、管理组内成员关系、管理元数据等。**ZooKeeper**包括一个应用开发库（主要提供Java和C两种语言的API）和一个用Java实现的服务组件。**ZooKeeper**的服务组件运行在一组专用服务器之上，保证了高容错性和可扩展性。

当你决定使用ZooKeeper来设计应用时，最好将应用数据和协同数据独立开。比如，网络邮箱服务的用户对自己邮箱中的内容感兴趣，但是并不关心由哪台服务器来处理特定邮箱的请求。在这个例子中，邮箱内容就是应用数据，而从邮箱到某一台邮箱服务器之间的映射关系就是协同数据（或称元数据）。整个ZooKeeper服务所管理的就是后者。

1.1 ZooKeeper的使命

试着说明ZooKeeper能为我们做什么，就像解释螺丝能为我们做什么一样。我们可以简单地表述，螺丝刀可以让我们拧动螺丝。但是这种方式并不能完全表达螺丝刀的能力。实际上，螺丝刀还可以让我们组装各种家具和电子设备，甚至在某些情况下你还可以用它把画挂在墙上。就像螺丝刀的例子一样，我们将介绍ZooKeeper能做什么，虽然未必详尽。

关于ZooKeeper这样的系统功能的讨论都围绕着一主线：它可以在分布式系统中协作多个任务。一个协作任务是指一个包含多个进程的任务。这个任务可以是为了协作或者是为了管理竞争。协作意味着多个进程需要一同处理某些事情，一些进程采取某些行动使得其他进程可以继续工作。比如，在典型的主-从（**master-worker**）工作模式中，从节点处于空闲状态时会通知主节点可以接受工作，于是主节点就会分配任务给从节点。竞争则不同。它指的是两个进程不能同时处理工作的情况，一个进程必须等待另一个进程。同样在主-从工作模式的例子中，我们想有一个主节点，但是很多进程也许都想成为主节点，因此我们需要实现互斥排他锁（**mutual exclusion**）。实际上，我们可以认为获取主节点身份的过程其实就是获取锁的过程，获得主节点控制权锁的进程即主节点进程。

如果你曾经有过多线程程序开发的经验，就会发现很多类似的问题。实际上，在一台计算机上运行的多个进程和跨计算机运行的多个进程从概念上区别并不大。在多线程情况下有用的同步原语在分布式系统中也同样有效。一个重要的区别在于，在典型的不共享环境下不同的计算机之间不共享除了网络之外的其他任何信息。虽然许多消息传递算法可以实现同步原语，但是使用一个提供某种有序共享存储的组件往往更加简便，这正是ZooKeeper所采用的方式。

协同并不总是采取像群首选举或者加锁等同步原语的形式。配置元数据也是一个进程通知其他进程需要做什么的一种常用方式。比如，在一个主-从系统中，从节点需要知道任务已经分配到它们。即使主节点发生崩溃的情况下，这些信息也需要有效。

让我们看一些ZooKeeper的使用实例，以便更直观地理解其用处：

Apache HBase

HBase是一个通常与Hadoop一起使用的数据存储仓库。在HBase中，ZooKeeper用于选举一个集群内的主节点，以便跟踪可用的服务器，并保存集群的元数据。

Apache Kafka

Kafka是一个基于发布-订阅（pub-sub）模型的消息系统。其中ZooKeeper用于检测崩溃，实现主题（topic）的发现，并保持主题的生产消费状态。

Apache Solr

Solr是一个企业级的搜索平台。Solr的分布式版本命名为SolrCloud，它使用ZooKeeper来存储集群的元数据，并协作更新这些元数据。

Yahoo! Fetching Service

Yahoo! Fetching Service是爬虫实现的一部分，通过缓存内容的方式高效地获取网页信息，同时确保满足网页服务器的管理规则（比如robots.txt文件）。该服务采用ZooKeeper实现主节点选举、崩溃检测和元数据存储。

Facebook Messages

Facebook推出的这个应用（<http://on.fb.me/1a7uViK>）集成了email、短信、Facebook聊天和Facebook收件箱等通信通道。该应用将ZooKeeper作为控制器，用来实现数据分片、故障恢复和服务发现等功能。

除了以上介绍的这些应用外，还有很多使用ZooKeeper的例子。根据这些代表应用，我们可以从更抽象的层次上讨论ZooKeeper。当开发人员使用ZooKeeper进行开发时，开发人员设计的那些应用往往可以看成一组连接到ZooKeeper服务器端的客户端，它们通过ZooKeeper的客户端API连接到ZooKeeper服务器端进行相应的操作。ZooKeeper的客户端API功能强大，其中包括：

- 保障强一致性、有序性和持久性。

- 实现通用的同步原语的能力。

- 在实际分布式系统中，并发往往导致不正确的行为。ZooKeeper提供了一种简单的并发处理机制。

当然，ZooKeeper也并不是万能的，我们还不能让它解决所有问题。对我们来说，更重要的是要了解ZooKeeper为我们提供了什么，并知道如何处理其中的一些棘手问题。这本书的目标之一就是讨论如何处理这些问题。我们将介绍ZooKeeper为开发人员提供的各种功能，也会讨论我们在开发ZooKeeper应用中遇到的一些问题，带你走入ZooKeeper的世界。

关于ZooKeeper名字的来源

ZooKeeper由雅虎研究院开发。我们小组在进行ZooKeeper的开发一段时间之后，开始推荐给其他小组，因此我们需要为我们的项目起一个名字。与此同时，小组也一同致力于Hadoop项目，参与了很多动物命名的项目，其中有广为人知的Apache Pig项目

(<http://pig.apache.org>)。我们在讨论各种各样的名字时，一位团队成员提到我们不能再使用动物的名字了，因为我们的主管觉得这样下去会觉得我们生活在动物园中。大家对此产生了共鸣，分布式系统就像一个动物园，混乱且难以管理，而ZooKeeper就是将这一切变得可控。

本书封面上的猫也颇有渊源。雅虎研究院早期的一篇关于ZooKeeper的文章谈到，分布式进程管理就像养一群猫一样，而ZooKeeper这个名字比CatHerder更好一些。

1.1.1 ZooKeeper改变了什么

使用ZooKeeper是否意味着需要以全新的方式进行应用程序开发？事实并非如此，ZooKeeper实际上简化了开发流程，提供了更加敏捷健壮的方案。

ZooKeeper之前的其他一些系统采用分布式锁管理器或者分布式数据库来实现协作。实际上，ZooKeeper也从这些系统中借鉴了很多概念。但是，ZooKeeper的设计更专注于任务协作，并不提供任何锁的接

口或通用存储数据接口。同时，ZooKeeper没有给开发人员强加任何特殊的同步原语，使用起来非常灵活。

虽然我们也可以不使用ZooKeeper来构建分布式系统，但是使用ZooKeeper可以让开发人员更专注于其应用本身的逻辑而不是神秘的分布式系统概念。所以不使用ZooKeeper开发分布式系统也并不是不可能，只是难度会比较大。

1.1.2 ZooKeeper不适用的场景

整个ZooKeeper的服务器集群管理着应用协作的关键数据。ZooKeeper不适合用作海量数据存储。对于需要存储海量的应用数据的情况，我们有很多备选方案，比如说数据库和分布式文件系统等。因为不同的应用有不同的需求，如对一致性和持久性的不同需求，所以在设计应用时，最佳实践还是应该将应用数据和协同数据独立开。

ZooKeeper中实现了一组核心操作，通过这些可以实现很多常见分布式应用的任务。你知道有多少应用服务采用主节点方式或进程响应跟踪方式？虽然ZooKeeper并没有为你实现这些任务，也没有为应用实现主节点选举，或者进程存活与否的跟踪的功能，但是，ZooKeeper提供了实现这些任务的工具，对于实现什么样的协同任务，由开发人员自己决定。

1.1.3 关于Apache项目

ZooKeeper是一个托管到Apache软件基金会（Apache Software Foundation）的开源项目，Apache的项目管理委员会（Project Management Committee, PMC）负责项目管理和监督。只有技术专家可以检查补丁（patch），但是任何开发人员都可以贡献补丁。开发人员为项目做出贡献后也可以成为技术专家。对项目的贡献不仅仅限于贡献补丁，也可以通过其他形式参与项目，与社区成员们互动。我们在邮件列表中进行很多讨论，如新功能特性、用户反馈的新问题等。我们强烈建议开发人员积极参与开发社区，订阅邮件列表，并参与讨论。如果你想通过一些项目与ZooKeeper社区保持长期关系，你也许会发现成为一名技术专家很有价值。

1.1.4 通过ZooKeeper构建分布式系统

对分布式系统的定义有很多，但对于本书的目的，我们对分布式系统的定义为：分布式系统是同时跨越多个物理主机，独立运行的多个软件组件所组成的系统。我们采用分布式去设计系统有很多原因，分布式系统能够利用多处理器的运算能力来运行组件，比如并行复制任务。一个系统也许由于战略原因，需要分布在不同地点，比如一个应用由多个不同地点的服务器提供服务。

使用一个独立的协调组件有几个重要的好处：首先，我们可以独立地设计和实现该组件，这样独立的组件可以跨多个应用共享。其次，系统架构师可以简化协作方面的工作，这些并不是琐碎的小事（本书试图说明这些）。最后，系统可以独立地运行和协作这些组件，独立这样的组件，也简化了生产环境中解决实际问题的任务。

软件组件以操作系统的进程方式运行，很多时候还涉及多线程的执行。因此ZooKeeper的服务端和客户端也是以进程的方式运行，一个单独的物理主机（无论是一个独立主机还是一个虚拟环境中的操作系统）上运行一个单独的应用进程，尽管进程可能采用多线程运行的方式，以便利用现代处理器的多核（multicore）处理能力。

分布式系统中的进程通信有两种选择：直接通过网络进行信息交换，或读写某些共享存储。ZooKeeper使用共享存储模型来实现应用间的协作和同步原语。对于共享存储本身，又需要在进程和存储间进行网络通信。我们强调网络通信的重要性，因为它是分布式系统中并发设计的基础。

在真实的系统中，我们需要特别注意以下问题：

消息延迟

消息传输可能会发生任意延迟，比如，因为网络拥堵。这种任意延迟可能会导致不可预期的后果。比如，根据基准时钟，进程P先发

送了一个消息，之后另一个进程Q发送了消息，但是进程Q的消息也许会先完成传送。

处理器性能

操作系统的调度和超载也可能导致消息处理的任意延迟。当一个进程向另一个进程发送消息时，整个消息的延时时间约等于发送端消耗的时间、传输时间、接收端的处理时间的总和。如果发送或接收过程需要调度时间进行处理，消息延时会更高。

时钟偏移

使用时间概念的系统并不少见，比如，确定某一时间系统中发生了哪些事件。处理器时钟并不可靠，它们之间也会发生任意的偏移。因此，依赖处理器时钟也许会导致错误的决策。

关于这些问题的一个重要结果是，在实际情况中，我们很难判断一个进程是崩溃了还是某些因素导致了延时。没有收到一个进程发送的消息，可能是该进程已经崩溃，或是最新消息发生了网络延迟，或是其他情况导致进程延迟，或者是进程时钟发生了偏移。我们无法确定一个被称为异步（asynchronous）的系统中的这些区别。

数据中心通常使用大量统一的硬件。但即使在数据中心，我们需要发现这些问题对应用服务带来的影响，因为一个应用服务使用了多

代的硬件，甚至对于同一批次的硬件也存在微小但显著的性能差异。所有这些事情使分布式系统设计师的生活越来越复杂。

ZooKeeper的精确设计简化了这些问题的处理，**ZooKeeper**并不是完全消除这些问题，而是将这些问题在应用服务层面上完全透明化，使得这些问题更容易处理。**ZooKeeper**实现了重要的分布式计算问题的解决方案，直观为开发人员提供某种程度上实现的封装，至少这是我们一直希望的。

1.2 示例：主-从应用

我们从理论上介绍了分布式系统，现在，是时候让它更具体一点了。考虑在分布式系统设计中一个得到广泛应用的架构：一个主-从（**master-worker**）架构（图1-1）。该系统中遵循这个架构的一个重要例子是HBase——一个Google的数据存储系统（**BigTable**）模型的实现，在最高层，主节点服务器（**HMaster**）负责跟踪区域服务器（**HRegionServer**）是否可用，并分派区域到服务器。因本书未涉及这些内容，如欲了解它如何使用**ZooKeeper**等更多细节，建议查看HBase相关文档。我们讨论的焦点是一般的主-从架构。

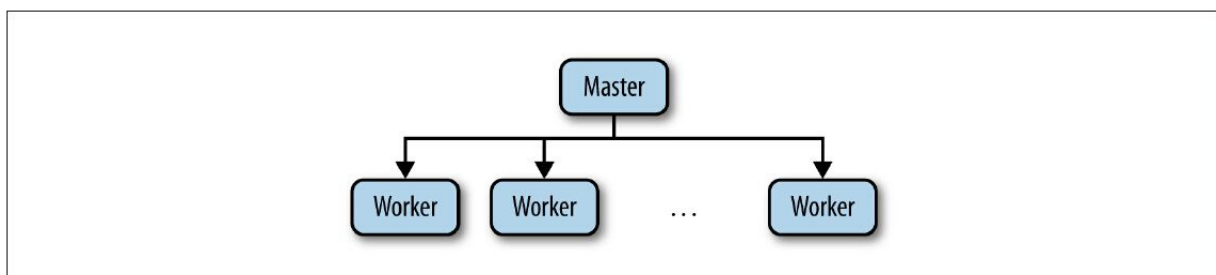


图1-1：主-从示例

一般在这种架构中，主节点进程负责跟踪从节点状态和任务的有效性，并分配任务到从节点。对**ZooKeeper**来说，这个架构风格具有代表性，阐述了大多数流行的任务，如选举主节点，跟踪有效的从节点，维护应用元数据。

要实现主-从模式的系统，我们必须解决以下三个关键问题：

主节点崩溃

如果主节点发送错误并失效，系统将无法分配新的任务或重新分配已失败的任务。

从节点崩溃

如果从节点崩溃，已分配的任务将无法完成。

通信故障

如果主节点和从节点之间无法进行信息交换，从节点将无法得知新任务分配给它。

为了处理这些问题，之前的主节点出现问题时，系统需要可靠地选举一个新的主节点，判断哪些从节点有效，并判定一个从节点的状态相对于系统其他部分是否失效。我们将会在下文中介绍这些任务。

1.2.1 主节点失效

主节点失效时，我们需要有一个备份主节点（**backup master**）。当主要主节点（**primary master**）崩溃时，备份主节点接管主要主节点的角色，进行故障转移，然而，这并不是简单开始处理进入主节点的请

求。新的主要主节点需要能够恢复到旧的主要主节点崩溃时的状态。对于主节点状态的可恢复性，我们不能依靠从已经崩溃的主节点来获取这些信息，而需要从其他地方获取，也就是通过ZooKeeper来获取。

状态恢复并不是唯一的重要问题。假如主节点有效，备份主节点却认为主节点已经崩溃。这种错误的假设可能发生在以下情况，例如主节点负载很高，导致消息任意延迟（关于这部分内容请参见1.1.4节），备份主节点将会接管成为主节点的角色，执行所有必需的程序，最终可能以主节点的角色开始执行，成为第二个主要主节点。更糟的是，如果一些从节点无法与主要主节点通信，如由于网络分区（**network partition**）错误导致，这些从节点可能会停止与主要主节点的通信，而与第二个主要主节点建立主-从关系。针对这个场景中导致的问题，我们一般称之为脑裂（**split-brain**）：系统中两个或者多个部分开始独立工作，导致整体行为不一致性。我们需要找出一种方法来处理主节点失效的情况，关键是我们需要避免发生脑裂的情况。

1.2.2 从节点失效

客户端向主节点提交任务，之后主节点将任务派发到有效的从节点中。从节点接收到派发的任务，执行完这些任务后会向主节点报告执行状态。主节点下一步会将执行结果通知给客户端。

如果从节点崩溃了，所有已派发给这个从节点且尚未完成的任务需要重新派发。其中首要需求是让主节点具有检测从节点的崩溃的能力。主节点必须能够检测到从节点的崩溃，并确定哪些从节点是否有效以便派发崩溃节点的任务。一个从节点崩溃时，从节点也许执行了部分任务，也许全部执行完，但没有报告结果。如果整个运算过程产生了其他作用，我们还有必要执行某些恢复过程来清除之前的状态。

1.2.3 通信故障

如果一个从节点与主节点的网络连接断开，比如网络分区（**network partition**）导致，重新分配一个任务可能会导致两个从节点执行相同的任务。如果一个任务允许多次执行，我们在进行任务再分配时可以不用验证第一个从节点是否完成了该任务。如果一个任务不允许，那么我们的应用需要适应多个从节点执行相同任务的可能性。

关于“仅一次”和“最多一次”的语义

对任务加锁并不能保证一个任务执行多次，比如以下场景中描述的情况：

- 1.主节点M1派发任务T1给从节点W1。
- 2.W1为任务T1获取锁，执行任务，然后释放锁。

3.M1怀疑W1已经崩溃，所以再次派发任务T1给从节点W2。

4.W2为任务T1获取锁，执行任务，然后释放锁。

在这里，T1的锁并没有阻止任务被执行两次，因为两个从节点间运行任务时没有步骤交错。处理类似情况就需要“仅一次”和“最多一次”的语义学，而这又依赖于应用的特定处理机制。例如，如果应用数据使用了时间戳数据，而假定任务会修改应用数据，那么该任务的执行成功就取决于这个任务所取得的这个时间戳的值。如果改变应用状态的操作不是原子性操作，那么应用还需要具有局部变更的回退能力，否则最终将导致应用的非一致性。

之所以讨论这些问题，最主要的原因是想说明实现这些语义学的应用是非常困难的。对这些语义学的实现细节并不是本书所讨论的内容。

通信故障导致的另一个重要问题是对锁等同步原语的影响。因为节点可能崩溃，而系统也可能网络分区（**network partition**），锁机制也会阻止任务的继续执行。因此ZooKeeper也需要实现处理这些情况的机制。首先，客户端可以告诉ZooKeeper某些数据的状态是临时状态（**ephemeral**）；其次，同时ZooKeeper需要客户端定时发送是否存活的通知，如果一个客户端未能及时发送通知，那么所有从属于这个客户

端的临时状态的数据将全部被删除。通过这两个机制，在崩溃或通信故障发生时，我们就可以预防客户端独立运行而发生的应用宕机。

回想一下之前讨论的内容，如果我们不能控制系统中的消息延迟，就不能确定一个客户端是崩溃还是运行缓慢，因此，当我们猜测一个客户端已经崩溃，而实际上我们也需要假设客户端仅仅是执行缓慢，其在后续还可能执行一些其他操作。

1.2.4 任务总结

根据之前描述的这些，我们可以得到以下主-从架构的需求：

主节点选举

这是关键的一步，使得主节点可以给从节点分配任务。

崩溃检测

主节点必须具有检测从节点崩溃或失去连接的能力。

组成员关系管理

主节点必须具有知道哪一个从节点可以执行任务的能力。

元数据管理

主节点和从节点必须具有通过某种可靠的方式来保存分配状态和执行状态的能力。

理想的方式是，以上每一个任务都需要通过原语的方式暴露给应用，对开发者完全隐藏实现细节。**ZooKeeper**提供了实现这些原语的关键机制，因此，开发者可以通过这些实现一个最适合他们需求、更加关注应用逻辑的分布式应用。贯穿本书，我们经常会涉及像主节点选举、崩溃检测这些原语任务的实现，因为这些是建立分布式应用的具体任务。

1.3 分布式协作的难点

当开发分布式应用时，其复杂性会立即突显出来。例如，当我们的应用启动后，所有不同的进程通过某种方法，需要知道应用的配置信息，一段时间之后，配置信息也许发生了变化，我们可以停止所有进程，重新分发配置信息的文件，然后重新启动，但是重新配置就会延长应用的停机时间。

与配置信息问题相关的是组成员关系的问题，当负载变化时，我们希望增加或减少新机器和进程。

当你自己实现分布式应用时，这个问题仅仅被描述为功能性问题，你可以设计解决方案，部署前你测试了你的解决方案，并非常确定地认为你已经正确解决了问题。当你在开发分布式应用时，你就会遇到真正困难的问题，你就不得不面对故障，如崩溃、通信故障等各种情况。这些问题会在任何可能的点突然出现，甚至无法列举需要处理的所有情况。

注意：拜占庭将军问题

拜占庭将军问题（**Byzantine Faults**）是指可能导致一个组件发生任意行为（常常是意料之外的）的故障。这个故障的组件可能会破坏

应用的状态，甚至是恶意行为。系统是建立在假设会发生这些故障，需要更高层次的复制并使用安全原语的基础上。尽管我们从学术文献中知道，针对拜占庭将军问题技术发展已经取得了巨大进步，我们还是觉得没有必要在ZooKeeper中采用这些技术，因此，我们也避免代码库中引入额外的复杂性。

在独立主机上运行的应用与分布式应用发生的故障存在显著的区别：在分布式应用中，可能会发生局部故障，当独立主机崩溃，这个主机上运行的所有进程都会失败，如果是独立主机上运行多个进程，一个进程执行的失败，其他进程可以通过操作系统获得这个故障，操作系统提供了健壮的多进程消息通信的保障。在分布式环境中这一切发生了改变：如果一个主机或进程发生故障，其他主机继续运行，并会接管发生故障的进程，为了能够处理故障进程，这些仍在运行的进程必须能够检测到这个故障，无论是消息丢失或发生了时间偏移。

理想的情况下，我们基于异步通信的假设来设计系统，即我们使用的主机有可能发生时间偏移或通信故障。我们做出这个假设是因为这一切的确会发生，时间偏移时常会发生，我们偶尔就会遇到网络问题，甚至更不幸的，发生故障。我们可以做什么样的限制呢？

我们来看一个最简单的情况。假设我们有一个分布式的配置信息发生了改变，这个配置信息简单到仅仅只有一个比特位（bit），一旦

所有运行中的进程对配置位的值达成一致，我们应用中的进程就可以启动。

这个例子原本是一个在分布式计算领域非常著名的定律，被称为FLP（由其作者命名：Fischer, Lynch, Patterson），这个结论证明了在异步通信的分布式系统中，进程崩溃，所有进程可能无法在这个比特位的配置上达成一致^[1]。类似的定律称为CAP，表示一致性

（Consistency）、可用性（Availability）和分区容错性（Partition-tolerance），该定律指出，当设计一个分布式系统时，我们希望这三种属性全部满足，但没有系统可以同时满足这三种属性^[2]。因此ZooKeeper的设计尽可能满足一致性和可用性，当然，在发生网络分区时ZooKeeper也提供了只读能力。

因此，我们无法拥有一个理想的故障容错的、分布式的、真实环境存在的系统来处理可能发生的所有问题。但我们还是可以争取一个稍微不那么宏伟的目标。首先，我们只好对我们的假设或目标适当放松，例如，我们可以假设时钟在某种范围内是同步的，我们也可以牺牲一些网络分区容错的能力并认为其一直是一致的，当一个进程运行中，也许多次因无法确定系统中的状态而被认为已经发生故障。虽然这些是一些折中方案，而这些折中方案允许我们建立一些印象非常深刻的分布式系统。

[1] Michael J.Fischer , Nancy A.Lynch , and Michael S.Paterson.“Impossibility of Distributed Consensus with One Faulty Process.”Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems , (1983) , doi : 10.1145/588058.588060.

[2] Seth Gilbert and Nancy Lynch.“Brewer’s Conjecture and the Feasibility of Consistent , Available , Partition-Tolerant Web Services.”ACM SIGACT News, 33: 2 (2002) , doi: 10.1145/564585.564601.

1.4 ZooKeeper的成功和注意事项

不得不指出，完美的解决方案是不存在的，我们重申ZooKeeper无法解决分布式应用开发者面对的所有问题，而是为开发者提供了一个优雅的框架来处理这些问题。多年以来，ZooKeeper在分布式计算领域进行了大量的工作。Paxos算法^[1]和虚拟同步技术（virtual synchrony）^[2]给ZooKeeper的设计带来了很大影响，通过这些技术可以无缝地处理所发生的某些变化或情况，并提供给开发者一个框架，来应对无法自动处理的某些情况。

ZooKeeper最初由雅虎研究院开发，用于处理大量的大型分布式应用。我们注意到，这些应用在分布式协作方面的处理方式并不妥当，这些系统的部署存在单点故障问题或很脆弱，另一方面，开发者在分布式协作方面花费了大量的时间和精力，导致开发者没有足够的资源来关注应用本身的功能逻辑。我们还注意到，这些应用都在基本协作方面有相同的需求。因此，我们开始着手设计一套通用的解决方案，通过某些关键点让我们可以一次实现就能应用于大多数不同的应用中。ZooKeeper已经被证实更加通用，其受欢迎程度超越了我们的想象。

多年来，我们发现人们可以很容易地部署ZooKeeper集群，轻松通过这个集群开发应用，但实际上，在使用ZooKeeper时，有些情况ZooKeeper自身无法进行决策而是需要开发者自己做出决策，有些开发者并不完全了解这些。编写本书的其中一个目的就是让开发者了解如何更有效地使用ZooKeeper，以及为什么需要这样做。

[1] Leslie Lamport.“The Part-Time Parliament.”ACM Transactions on Computer Systems, 16: 2 (1998) : 133–169.

[2] K.Birman and T.Joseph.“Exploiting Virtual Synchrony in Distributed Systems.”Proceedings of the 11th ACM Symposium on Operating Systems Principles, (1987) : 123–138.

第2章 了解ZooKeeper

前一章从较高的层面讨论了分布式应用的需求，同时也讨论了在协作方面的共性需求。我们以实际应用中使用的很广泛的主-从架构（**master-worker**）为例子，从中摘取了一些常用原语。本章将开始讨论ZooKeeper，看一看这个服务如何实现这些协作方面的原语。

2.1 ZooKeeper基础

很多用于协作的原语常常在很多应用之间共享，因此，设计一个用于协作需求的服务的方法往往是提供原语列表，暴露出每个原语的实例化调用方法，并直接控制这些实例。比如，我们可以说分布式锁机制组成了一个重要的原语，同时暴露出创建（create）、获取（acquire）和释放（release）三个调用方法。

这种设计存在一些重大的缺陷：首先，我们要么预先提出一份详尽的原语列表，要么提供API的扩展，以便引入新的原语；其次，以这种方式实现原语的服务使得应用丧失了灵活性。

因此，在ZooKeeper中我们另辟蹊径。ZooKeeper并不直接暴露原语，取而代之，它暴露了由一小部分调用方法组成的类似文件系统的API，以便允许应用实现自己的原语。我们通常使用菜谱（recipes）来表示这些原语的实现。菜谱包括ZooKeeper操作和维护一个小型的数据节点，这些节点被称为znode，采用类似于文件系统的层级树状结构进行管理。图2-1描述了一个znode树的结构，根节点包含4个子节点，其中三个子节点拥有下一级节点，叶子节点存储了数据信息。

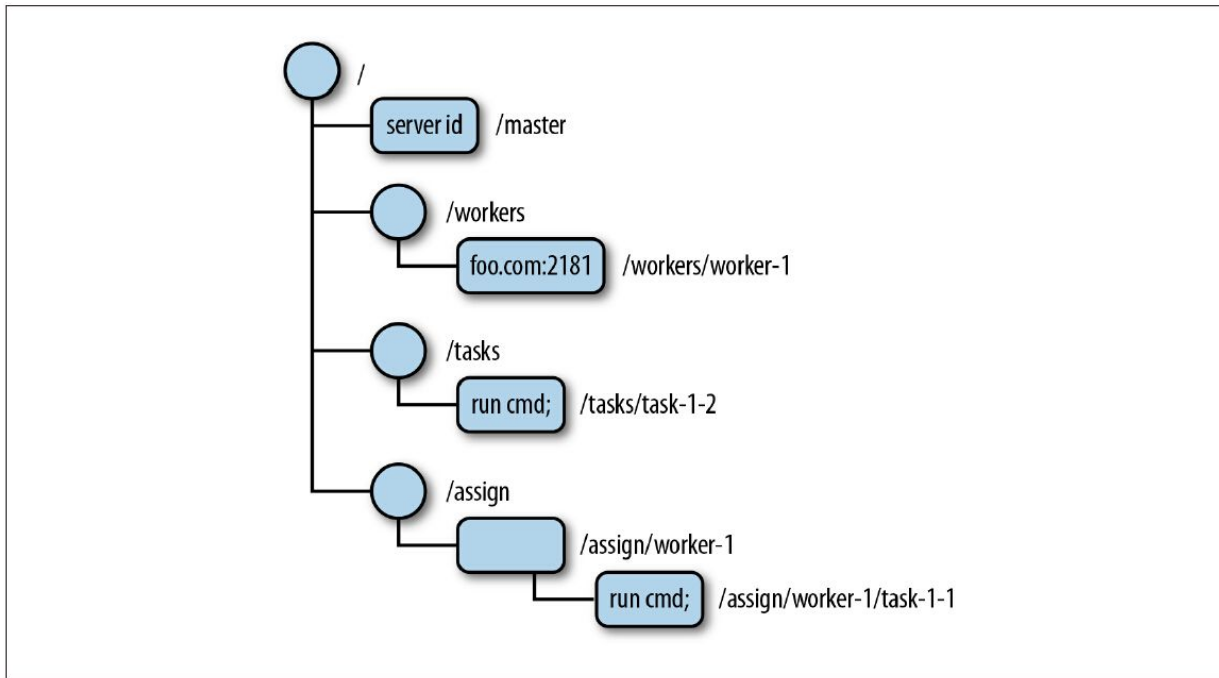


图2-1: ZooKeeper数据树结构示例

针对一个znode，没有数据常常表达了重要的信息。比如，在主-从模式的例子中，主节点的znode没有数据，表示当前还没有选举出主节点。而图2-1中涉及的一些其他znode节点在主-从模式的配置中非常有用：

- /workers节点作为父节点，其下每个znode子节点保存了系统中一个可用从节点信息。如图2-1所示，有一个从节点（foo.com:2181）。

- /tasks节点作为父节点，其下每个znode子节点保存了所有已经创建并等待从节点执行的任务的信息，主-从模式的应用的客户端在/tasks

下添加一个znode子节点，用来表示一个新任务，并等待任务状态的znode节点。

·/assign节点作为父节点，其下每个znode子节点保存了分配到某个从节点的一个任务信息，当主节点为某个从节点分配了一个任务，就会在/assign下增加一个子节点。

2.1.1 API概述

znode节点可能含有数据，也可能没有。如果一个znode节点包含任何数据，那么数据存储为字节数组（byte array）。字节数组的具体格式特定于每个应用的实现，ZooKeeper并不直接提供解析的支持。我们可以使用如Protocol Buffers、Thrift、Avro或MessagePack等序列化

（Serialization）包来方便地处理保存于znode节点的数据格式，不过有些时候，以UTF-8或ASCII编码的字符串已经够用了。

ZooKeeper的API暴露了以下方法：

create/path data

创建一个名为/path的znode节点，并包含数据data。

delete/path

删除名为/path的znode。

exists/path

检查是否存在名为/path的节点。

setData/path data

设置名为/path的znode的数据为data。

getData/path

返回名为/path节点的数据信息。

getChildren/path

返回所有/path节点的所有子节点列表。

需要注意的是，ZooKeeper并不允许局部写入或读取znode节点的数据。当设置一个znode节点的数据或读取时，znode节点的内容会被整个替换或全部读取进来。

ZooKeeper客户端连接到ZooKeeper服务，通过API调用来建立会话（session）。如果你对如何使用ZooKeeper非常感兴趣，请跳转到后面的“会话”一节，在那一节会讲解如何通过命令行的方式来运行ZooKeeper指令。

2.1.2 znode的不同类型

当新建znode时，还需要指定该节点的类型（mode），不同的类型决定了znode节点的行为方式。

持久节点和临时节点

znode节点可以是持久（persistent）节点，还可以是临时（ephemeral）节点。持久的znode，如/path，只能通过调用delete来进行删除。临时的znode与之相反，当创建该节点的客户端崩溃或关闭了与ZooKeeper的连接时，这个节点就会被删除。

持久znode是一种非常有用的znode，可以通过持久类型的znode为应用保存一些数据，即使znode的创建者不再属于应用系统时，数据也可以保存下来而不丢失。例如，在主-从模式例子中，需要保存从节点的任务分配情况，即使分配任务的主节点已经崩溃了。

临时znode传达了应用某些方面的信息，仅当创建者的会话有效时这些信息必须有效保存。例如，在主从模式的例子中，当主节点创建的znode为临时节点时，该节点的存在意味着现在有一个主节点，且主节点状态处于正常运行中。如果主znode消失后，该znode节点仍然存在，那么系统将无法监测到主节点崩溃。这样就可以阻止系统继续进行，因此这个znode需要和主节点一起消失。我们也在从节点中使用临时的znode，如果一个从节点失效，那么会话将会过期，之后znode/workers也将自动消失。

一个临时znode，在以下两种情况下将会被删除：

- 1.当创建该znode的客户端的会话因超时或主动关闭而中止时。
- 2.当某个客户端（不一定是创建者）主动删除该节点时。

因为临时的znode在其创建者的会话过期时被删除，所以我们现在不允许临时节点拥有子节点。在社区讨论中，已经讨论过关于允许临时znode拥有子节点的问题，其想法是使其子节点也均为临时节点。这个功能也许会出现在未来的发布版本中，但现在还是不可用的。

有序节点

一个znode还可以设置为有序（sequential）节点。一个有序znode节点被分配唯一一个单调递增的整数。当创建有序节点时，一个序号会被追加到路径之后。例如，如果一个客户端创建了一个有序znode节点，其路径为/tasks/task-，那么ZooKeeper将会分配一个序号，如1，并将这个数字追加到路径之后，最后该znode节点为/tasks/task-1。有序znode通过提供了创建具有唯一名称的znode的简单方式。同时也通过这种方式可以直观地查看znode的创建顺序。

总之，znode一共有4种类型：持久的（persistent）、临时的（ephemeral）、持久有序的（persistent_sequential）和临时有序的（ephemeral_sequential）。

2.1.3 监视与通知

ZooKeeper通常以远程服务的方式被访问，如果每次访问znode时，客户端都需要获得节点中的内容，这样的代价就非常大。因为这样会导致更高的延迟，而且ZooKeeper需要做更多的操作。考虑图2-2中的例子，第二次调用getChildren/tasks返回了相同的值，一个空的集合，其实是没有必要的。

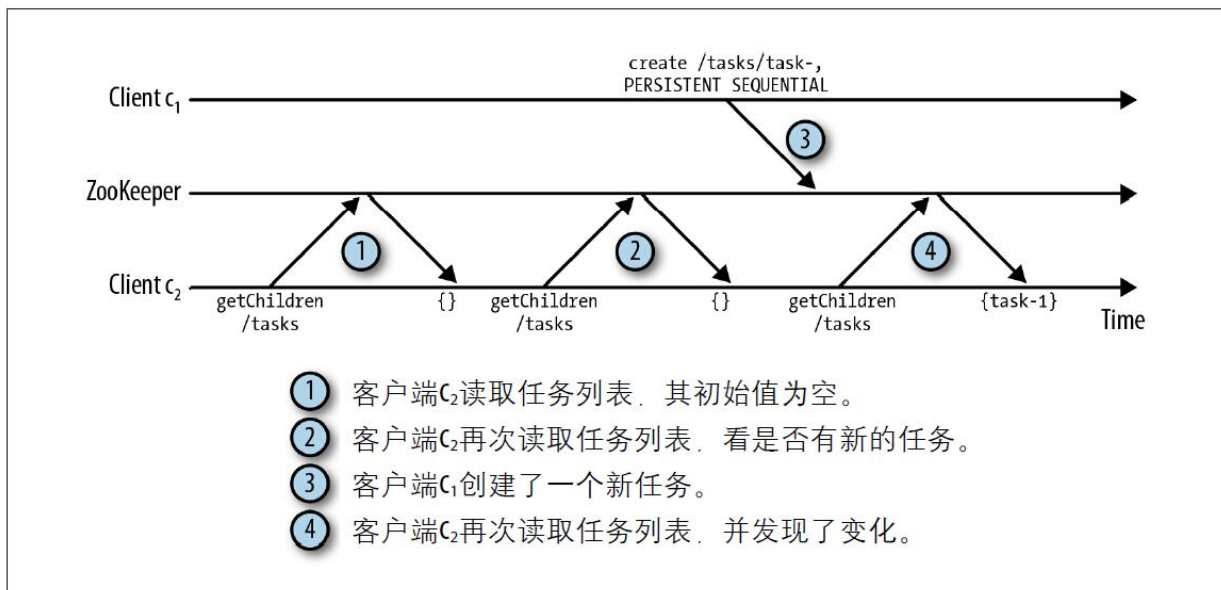


图2-2: 同一个znode的多次读取

这是一个常见的轮询问题。为了替换客户端的轮询，我们选择了基于通知（notification）的机制：客户端向ZooKeeper注册需要接收通知的znode，通过对znode设置监视点（watch）来接收通知。监视点是一个单次触发的操作，意即监视点会触发一个通知。为了接收多个通知，客户端必须在每次通知后设置一个新的监视点。在图2-3阐述的情

况下，当节点/tasks发生变化时，客户端会收到一个通知，并从 ZooKeeper 读取一个新值。

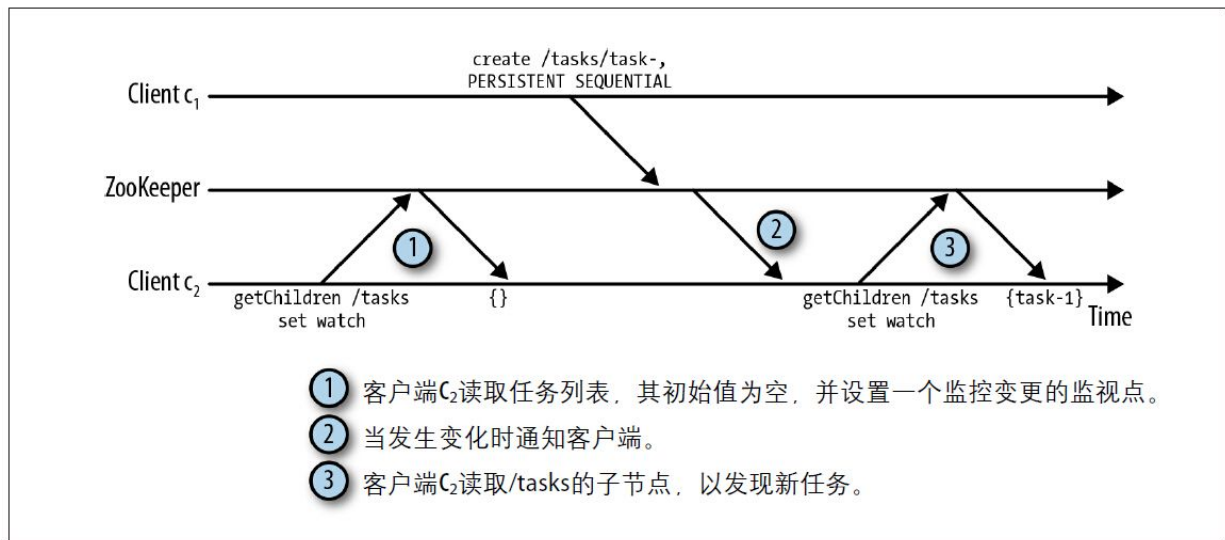


图2-3：使用通知机制来获悉znode的变化

当使用通知机制时，还有一些需要知道的事情。因为通知机制是单次触发的操作，所以在客户端接收一个 **znode** 变更通知并设置新的监视点时，**znode** 节点也许发生了新的变化（不要担心，你不会错过状态的变化）。让我们看一个例子来说明它到底是怎么工作的。假设事件按以下顺序发生：

- 1.客户端 c_1 设置监视点来监控/tasks数据的变化。
- 2.客户端 c_1 连接后，向/tasks中添加了一个新的任务。
- 3.客户端 c_1 接收通知。

4.客户端 c_1 设置新的监视点，在设置完成前，第三个客户端 c_3 连接后，向/tasks中添加了一个新的任务。

客户端 c_1 最终设置了新的监视点，但由 c_3 添加数据的变更并没有触发一个通知。为了观察这个变更，在设置新的监视点前， c_1 实际上需要读取节点/tasks的状态，通过在设置监视点前读取ZooKeeper的状态，最终， c_1 就不会错过任何变更。

通知机制的一个重要保障是，对同一个znode的操作，先向客户端传送通知，然后再对该节点进行变更。如果客户端对一个znode设置了监视点，而该znode发生了两个连续更新。第一次更新后，客户端在观察第二次变化前就接收到了通知，然后读取znode中的数据。我们认为主要特性在于通知机制阻止了客户端所观察的更新顺序。虽然ZooKeeper的状态变化传播给某些客户端时更慢，但我们保障客户端以全局的顺序来观察ZooKeeper的状态。

ZooKeeper可以定义不同类型的通知，这依赖于设置监视点对应的通知类型。客户端可以设置多种监视点，如监控znode的数据变化、监控znode子节点的变化、监控znode的创建或删除。为了设置监视点，可以使用任何API中的调用来读取ZooKeeper的状态，在调用这些API时，传入一个watcher对象或使用默认的watcher。本章后续（主从模式的实现）及第4章会以主从模式的例子来展开讨论，我们将深入研究如何使用该机制。

注意：谁来管理我的缓存

如果不让客户端来管理其拥有的ZooKeeper数据的缓存，我们不得不让ZooKeeper来管理这些应用程序的缓存。但是，这样会导致ZooKeeper的设计更加复杂。事实上，如果让ZooKeeper管理缓存失效，可能会导致ZooKeeper在运行时，停滞在等待客户端确认一个缓存失效的请求上，因为在进行所有的写操作前，需要确认所有的缓存数据是否已经失效。

2.1.4 版本

每一个znode都有一个版本号，它随着每次数据变化而自增。两个API操作可以有条件地执行：`setData`和`delete`。这两个调用以版本号作为转入参数，只有当转入参数的版本号与服务器上的版本号一致时调用才会成功。当多个ZooKeeper客户端对同一个znode进行操作时，版本的使用就会显得尤为重要。例如，假设客户端 c_1 对znode/config写入了一些配置信息，如果另一个客户端 c_2 同时更新了这个znode，此时 c_1 的版本号已经过期， c_1 调用`setData`一定不会成功。使用版本机制有效避免了以上情况。在这个例子中， c_1 在写入数据时使用的版本无法匹配，使得操作失败，图2-4描述了这个情况。

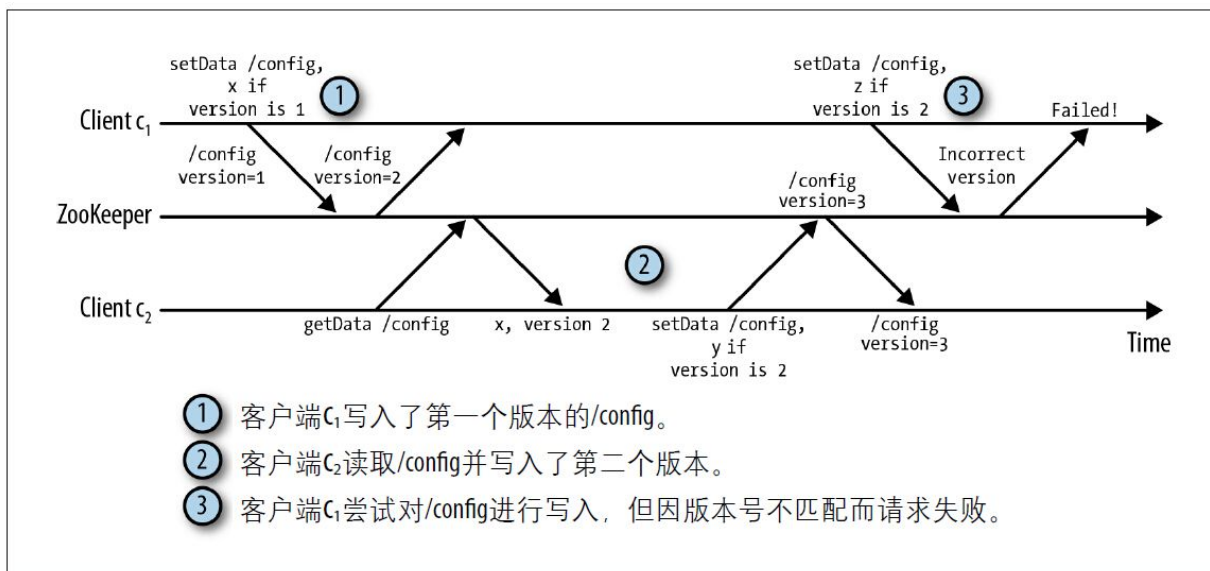


图2-4：使用版本来阻止并行操作的不一致性

2.2 ZooKeeper架构

现在我们已经讨论了ZooKeeper暴露给应用的高层操作，我们需要详细了解服务实际上是如何运行的。应用通过客户端库来对ZooKeeper实现了调用。客户端库负责与ZooKeeper服务器端进行交互。

图2-5展示了客户端与服务器端之间的关系。每一个客户端导入客户端库，之后便可以与任何ZooKeeper的节点进行通信。

ZooKeeper服务器端运行于两种模式下：独立模式（standalone）和仲裁模式（quorum）。独立模式几乎与其术语所描述的一样：有一个单独的服务器，ZooKeeper状态无法复制。在仲裁模式下，具有一组ZooKeeper服务器，我们称为ZooKeeper集合（ZooKeeper ensemble），它们之前可以进行状态的复制，并同时为服务于客户端的请求。从这个角度出发，我们使用术语“ZooKeeper集合”来表示一个服务器设施，这一设施可以由独立模式的一个服务器组成，也可以仲裁模式下的多个服务器组成。

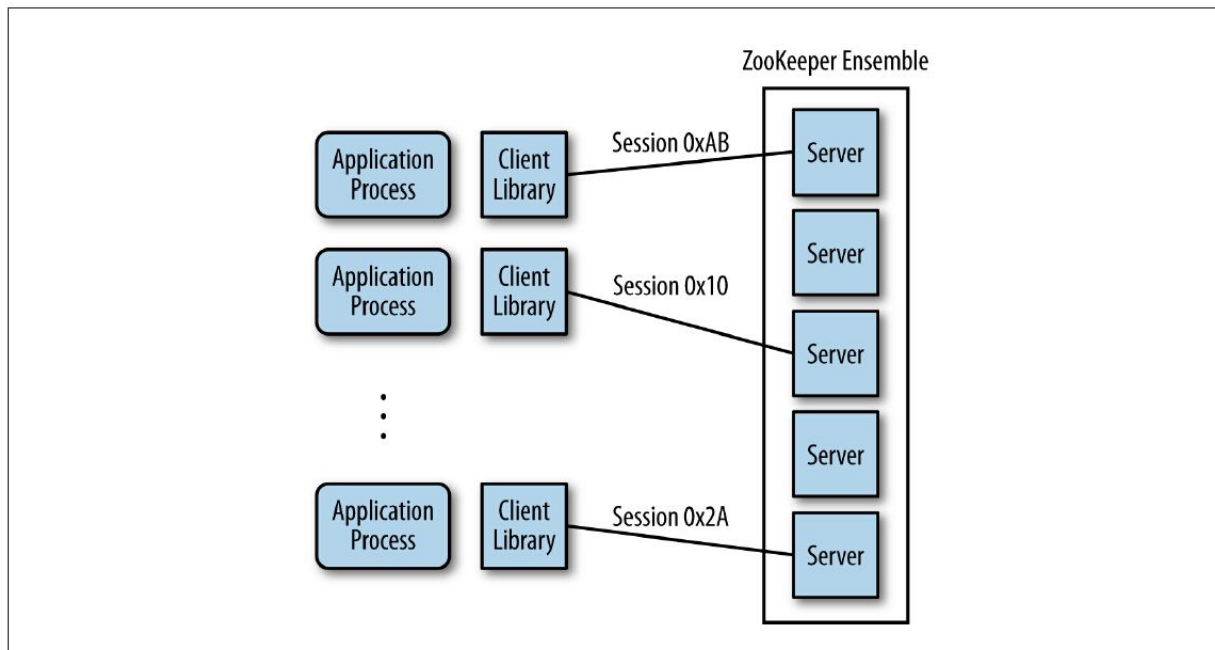


图2-5: ZooKeeper架构总览

2.2.1 ZooKeeper仲裁

在仲裁模式下，ZooKeeper复制集群中的所有服务器的数据树。但如果让一个客户端等待每个服务器完成数据保存后再继续，延迟问题将无法接受。在公共管理领域，法定人数是指进行一项投票所需的立法者的最小数量。而在ZooKeeper中，则是指为了使ZooKeeper工作必须有效运行的服务器的最小数量。这个数字也是服务器告知客户端安全保存数据前，需要保存客户端数据的服务器的最小个数。例如，我们一共有5个ZooKeeper服务器，但法定人数为3个，这样，只要任何3个服务器保存了数据，客户端就可以继续，而其他两个服务器最终也将捕获到数据，并保存数据。

选择法定人数准确的大小是一个非常重要的事。法定人数的数量需要保证不管系统发生延迟或崩溃，服务主动确认的任何更新请求需要保持下去，直到另一个请求代替它。

为了明白这到底是什么意思，让我们先来通过一个例子来看看，如果法定人数太小，会如何出错。假设有5个服务器并设置法定人数为2，现在服务器 s_1 和 s_2 确认它们需要对一个请求创建的`znode/z`进行复制，服务返回客户端，指出`znode`创建完成。现在假设在复制新的`znode`到其他服务器之前，服务器 s_1 和 s_2 与其他服务器和客户端发生了长时间的分区隔离，整个服务的状态仍然正常，因为基于我们的假设设定法定人数为2，而现在还有3个服务器，但这3个服务器将无法发现新的`znode/z`。因此，对创建节点`/z`的请求是非持久化的。

这就是第1章中讲述的脑裂场景的例子。为了避免这个问题，这个例子中，法定人数的大小必须至少为3，即集合中5个服务器的多数原则。为了能正常工作，集合中至少要有3个有效的服务器。为了确认一个请求对状态的更新是否成功完成，这个集合同时需要至少3个服务器确认已经完成了数据的复制操作。因此，如果要保证集合可以正常工作，对任何更新操作的成功完成，我们至少要有1个有效的服务器来保存更新的副本（即至少在一个节点上合理的法定人数存在交集）。

通过使用多数方案，我们就可以容许 f 个服务器的崩溃，在这里， f 为小于集合中服务器数量的一半。例如，如果有5个服务器，可以容许

最多 $f=2$ 个崩溃。在集合中，服务器的个数并不是必须为奇数，只是使用偶数会使得系统更加脆弱。假设在集合中使用4个服务器，那么多数原则对应的数量为3个服务器。然而，这个系统仅能容许1个服务器崩溃，因为两个服务器崩溃就会导致系统失去多数原则的状态。因此，在4个服务器的情况下，我们仅能容许一个服务器崩溃，而法定人数现在却更大，这意味着对每个请求，我们需要更多的确认操作。底线是我们需要争取奇数个服务器。

我们允许法定人数的数量不同于多数原则，但这将在后续章节深入讨论。第10章会讨论此问题。

2.2.2 会话

在对ZooKeeper集合执行任何请求前，一个客户端必须先与服务建立会话。会话的概念非常重要，对ZooKeeper的运行也非常关键。客户端提交给ZooKeeper的所有操作均关联在一个会话上。当一个会话因某种原因而中止时，在这个会话期间创建的临时节点将会消失。

当客户端通过某一个特定语言套件来创建一个ZooKeeper句柄时，它就会通过服务建立一个会话。客户端初始连接到集合中某一个服务器或一个独立的服务器。客户端通过TCP协议与服务器进行连接并通信，但当会话无法与当前连接的服务器继续通信时，会话就可能转移

到另一个服务器上。ZooKeeper客户端库透明地转移一个会话到不同的服务器。

会话提供了顺序保障，这就意味着同一个会话中的请求会以FIFO（先进先出）顺序执行。通常，一个客户端只打开一个会话，因此客户端请求将全部以FIFO顺序执行。如果客户端拥有多个并发的会话，FIFO顺序在多个会话之间未必能够保持。而即使一个客户端中连贯的会话并不重叠，也未必能够保证FIFO顺序。下面的情况说明如何发生这种问题：

- 客户端建立了一个会话，并通过两个连续的异步调用来创建/tasks和/workers。

- 第一个会话过期。

- 客户端创建另一个会话，并通过异步调用创建/assign。

在这个调用顺序中，可能只有/tasks和/assign成功创建了，因为第一个会话保持了FIFO顺序，但在跨会话时就违反了FIFO顺序。

2.3 开始使用ZooKeeper

开始之前，需要下载ZooKeeper发行包。ZooKeeper作为Apache项目托管到<http://zookeeper.apache.org>。通过下载链接，你会下载到一个名字类似zookeepe-3.4.5.tar.gz的压缩TAR格式文件。在Linux、Mac OS X或任何其他类UNIX系统上，可以通过一下命令解压缩发行包：

```
# tar -xvzf zookeeper-3.4.5.tar.gz
```

如果使用Windows，可以使用如WinZip等解压缩工具来解压发行包。

在发行包（distribution）的目录中，你会发现在bin目录中有启动ZooKeeper的脚本。以.sh结尾的脚本运行于UNIX平台（Linux、Mac OS X等），以.cmd结尾的脚本则用于Windows。在conf目录中保存配置文件。lib目录包括Java的JAR文件，它们是运行ZooKeeper所需要的第三方文件。稍后我们需要引用ZooKeeper解压缩的目录。我们以{PATH_TO_ZK}方式来引用该目录。

2.3.1 第一个ZooKeeper会话

首先我们以独立模式运行ZooKeeper并创建一个会话。要做到这一点，使用ZooKeeper发行包中bin/目录下的zkServer和zkCli工具。有经验的管理人员常常使用这两个工具来进行调试和管理，同时也非常适合初学者熟悉和了解ZooKeeper。

假设你已经下载并解压了ZooKeeper发行包，进入shell，变更目录（cd）到项目根目录下，重命名配置文件：

```
# mv conf/zoo_sample.cfg conf/zoo.cfg
```

虽然是可选的，最好还是把data目录移出/tmp目录，以防止ZooKeeper填满了根分区（root partition）。可以在zoo.cfg文件中修改这个目录的位置。

```
dataDir=/users/me/zookeeper
```

最后，为了启动服务器，执行以下命令：

```
# bin/zkServer.sh start
JMX enabled by default
Using config: ../conf/zoo.cfg
Starting zookeeper ... STARTED
#
```

这个服务器命令使得ZooKeeper服务器在后台中运行。如果在前台中运行以便查看服务器的输出，可以通过以下命令运行：

```
# bin/zkServer.sh start-foreground
```

这个选项提供了大量详细信息的输出，以便允许查看服务器发生了什么。

现在我们准备启动客户端。在另一个shell中进入项目根目录，运行以下命令：

```
# bin/zkCli.sh
.
.
.
<some omitted output>
.
.
.
2012-12-06 12:07:23,545 [myid:] - INFO [main:ZooKeeper@438] - ①
```

```
Initiating client connection, connectString=localhost:2181
sessionTimeout=30000 watcher=org.apache.zookeeper.
ZooKeeperMain$MyWatcher@2c641e9a
Welcome to ZooKeeper!
2012-12-06 12:07:23,702 [myid:] - INFO [main-SendThread②
```

```
(localhost:2181):ClientCnxn$SendThread@966] - Opening
socket connection to server localhost/127.0.0.1:2181.
Will not attempt to authenticate using SASL (Unable to
locate a login configuration)
JLine support is enabled
2012-12-06 12:07:23,717 [myid:] - INFO [main-SendThread③
```

```
(localhost:2181):ClientCnxn$SendThread@849] - Socket
connection established to localhost/127.0.0.1:2181, initiating
session [zk: localhost:2181(CONNECTING) 0]
2012-12-06 12:07:23,987 [myid:] - INFO [main-SendThread④
```

```
(localhost:2181):ClientCnxn$SendThread@1207] - Session
```



```
establishment complete on server localhost/127.0.0.1:2181,  
sessionid = 0x13b6fe376cd0000, negotiated timeout = 30000  
WATCHER::  
WatchedEvent state:SyncConnected type:None path:null⑤
```

- ①客户端启动程序来建立一个会话。
- ②客户端尝试连接到localhost/127.0.0.1: 2181。
- ③客户端连接成功，服务器开始初始化这个新会话。
- ④会话初始化成功完成。
- ⑤服务器向客户端发送一个SyncConnected事件。

让我们来看一看这些输出。有很多行告诉我们各种各样的环境变量的配置以及客户端使用了什么JAR包。我们在例子中忽略这些信息，关注会话的建立，但你可以花时间来分析所有屏幕的输出信息。

在输出的结尾，我们看到会话建立的日志消息。第一处提到“Initiating client connection.”。消息本身说明到底发生了什么，而额外的重要细节说明了客户端尝试连接到客户端发送的连接串localhost/127.0.0.1: 2181中的一个服务器。这个例子中，字符串只包含了localhost，因此指明了具体连接的地址。之后我们看到关于SASL的消息，我们暂时忽略这个消息，随后一个确认信息说明客户端与本地

的ZooKeeper服务器建立了TCP连接。后面的日志信息确认了会话的建立，并告诉我们会话ID为：0x13b6fe376cd0000。最后客户端库通过SyncConncted事件通知了应用。应用需要实现Watcher对象来处理这个事件。下一节将详细说明事件。

为了更加了解ZooKeeper，让我们列出根（root）下的所有znode，然后创建一个znode。首先我们要确认此刻znode树为空，除了节点/zookeeper之外，该节点内标记了ZooKeeper服务所需的元数据树。

```
WATCHER::
WatchedEvent state:SyncConnected type:None path:null
[zk: localhost:2181(CONNECTED) 0] ls /
[zookeeper]
```

现在发生了什么？我们执行ls/后看到这里只有/zookeeper节点。现在我们创建一个名为/workers的znode，确保如下所示：

```
WATCHER::
WatchedEvent state:SyncConnected type:None path:null
[zk: localhost:2181(CONNECTED) 0]
[zk: localhost:2181(CONNECTED) 0] ls /
[zookeeper]
[zk: localhost:2181(CONNECTED) 1] create /workers ""
Created /workers
[zk: localhost:2181(CONNECTED) 2] ls /
[workers, zookeeper]
[zk: localhost:2181(CONNECTED) 3]
```

注意：Znode数据

当创建/workers节点后，我们指定了一个空字符串（""），说明我们此刻不希望在这个znode中保存数据。然而，该接口中的这个参数可

以使我们保存任何字符串到ZooKeeper的节点中。比如，可以替换""为"workers"。

为了完成这个练习，删除znode，然后退出：

```
[zk: localhost:2181(CONNECTED) 3] delete /workers
[zk: localhost:2181(CONNECTED) 4] ls /
[zookeeper]
[zk: localhost:2181(CONNECTED) 5] quit
Quitting...
2012-12-06 12:28:18,200 [myid:] - INFO [main-EventThread:ClientCnxn$
EventThread@509] - EventThread shut down
2012-12-06 12:28:18,200 [myid:] - INFO [main:ZooKeeper@684] - Session:
0x13b6fe376cd0000 closed
```

观察到znode/workers已经被删除，并且会话现在也关闭。为了完成最后的清理，退出ZooKeeper服务器：

```
# bin/zkServer.sh stop
JMX enabled by default
Using config: ../conf/zoo.cfg
Stopping zookeeper ... STOPPED
#
```

2.3.2 会话的状态和声明周期

会话的生命周期（lifetime）是指会话从创建到结束的时期，无论会话正常关闭还是因超时而导致过期。为了讨论在会话中发生了什么，我们需要考虑会话可能的状态，以及可能导致会话状态改变的事件。

一个会话的主要可能状态大多是简单明了的：CONNECTING、CONNECTED、CLOSED和NOT_CONNECTED。状态的转换依赖于发生在客户端与服务之间的各种事件（见图2-6）。

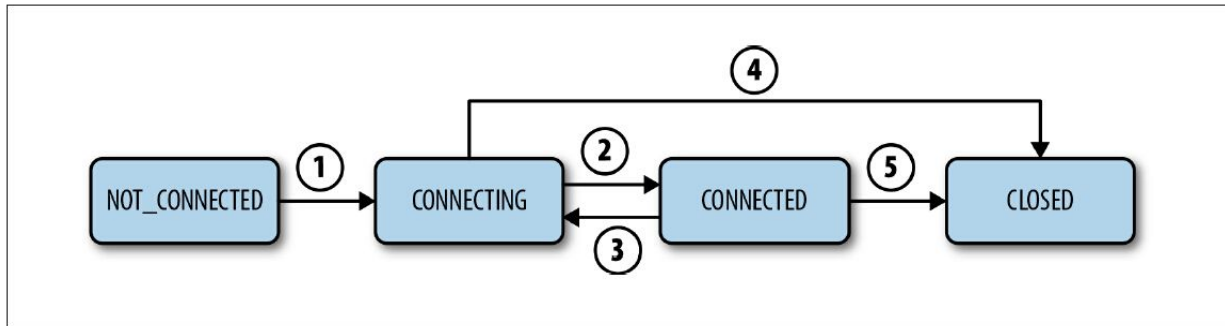


图2-6：状态及状态的转换

一个会话从NOT_CONNECTED状态开始，当ZooKeeper客户端初始化后转换到CONNECTING状态（图2-6中的箭头1）。正常情况下，成功与ZooKeeper服务器建立连接后，会话转换到CONNECTED状态（箭头2）。当客户端与ZooKeeper服务器断开连接或者无法收到服务器的响应时，它就会转换回CONNECTING状态（箭头3）并尝试发现其他ZooKeeper服务器。如果可以发现另一个服务器或重连到原来的服务器，当服务器确认会话有效后，状态又会转换回CONNECTED状态。否则，它将会声明会话过期，然后转换到CLOSED状态（箭头4）。应用也可以显式地关闭会话（箭头4和箭头5）。

注意： 发生网络分区时等待CONNECTING

如果一个客户端与服务器因超时而断开连接，客户端仍然保持CONNECTING状态。如果因网络分区问题导致客户端与ZooKeeper集合被隔离而发生连接断开，那么其状态将会一直保持，直到显式地关闭这个会话，或者分区问题修复后，客户端能够获悉ZooKeeper服务器发送的会话已经过期。发生这种行为是因为ZooKeeper集合对声明会话超时负责，而不是客户端负责。直到客户端获悉ZooKeeper会话过期，否则客户端不能声明自己的会话过期。然而，客户端可以选择关闭会话。

创建一个会话时，你需要设置会话超时这个重要的参数，这个参数设置了ZooKeeper服务允许会话被声明为超时之前存在的时间。如果经过时间 t 之后服务接收不到这个会话的任何消息，服务就会声明会话过期。而在客户端侧，如果经过 $t/3$ 的时间未收到任何消息，客户端将向服务器发送心跳消息。在经过 $2t/3$ 时间后，ZooKeeper客户端开始寻找其他的服务器，而此时它还有 $t/3$ 时间去寻找。

注意：客户端会尝试连接哪一个服务器？

在仲裁模式下，客户端有多个服务器可以连接，而在独立模式下，客户端只能尝试重新连接单个服务器。在仲裁模式中，应用需要传递可用的服务器列表给客户端，告知客户端可以连接的服务器信息并选择一个进行连接。

当尝试连接到一个不同的服务器时，非常重要的一点是，这个服务器的ZooKeeper状态要与最后连接的服务器的ZooKeeper状态保持最新。客户端不能连接到这样的服务器：它未发现更新而客户端却已经发现的更新。ZooKeeper通过在服务中排序更新操作来决定状态是否最新。ZooKeeper确保每一个变化相对于所有其他已执行的更新是完全有序的。因此，如果一个客户端在位置 i 观察到一个更新，它就不能连接到只观察到 $i' < i$ 的服务器上。在ZooKeeper实现中，系统根据每一个更新建立的顺序来分配给事务标识符。

图2-7描述了在重连情况下事务标识符（zkid）的使用。当客户端因超时与 s_1 断开连接后，客户端开始尝试连接 s_2 ，但 s_2 延迟于客户端所知的变化。然而， s_3 对这个变化的情况与客户端保持一致，所以 s_3 可以安全连接。

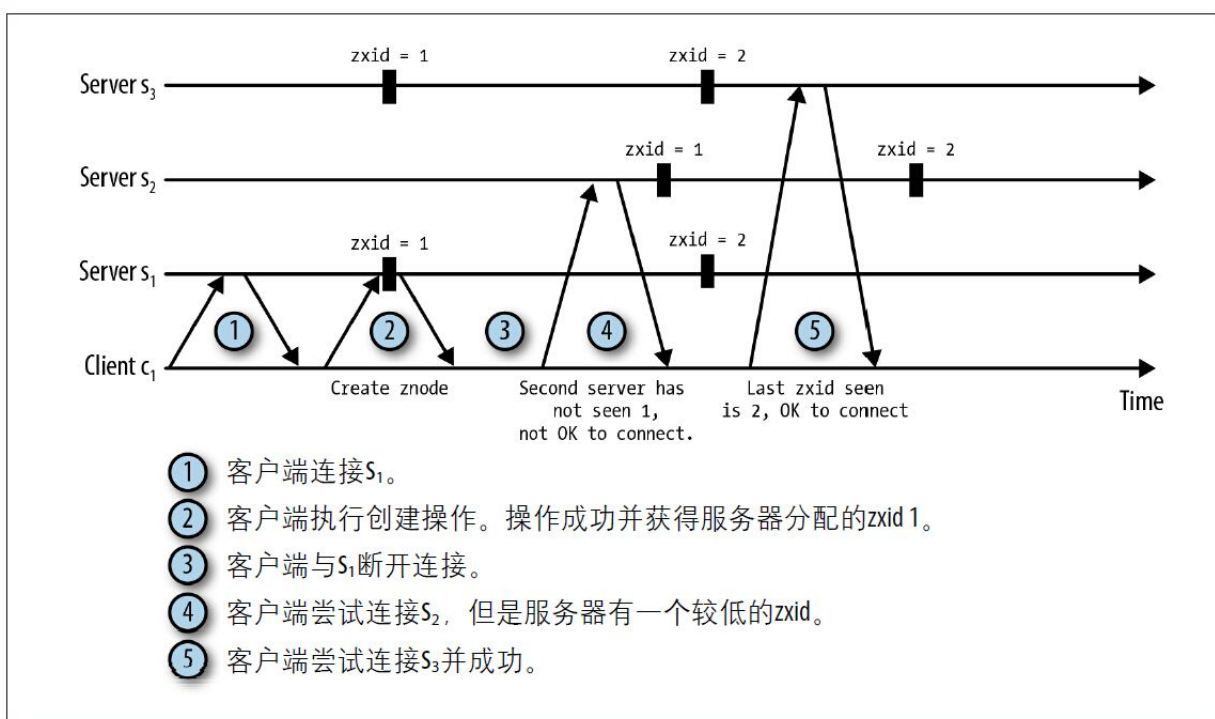


图2-7：客户端重连的例子

2.3.3 ZooKeeper与仲裁模式

到目前为止，我们一直基于独立模式配置的服务器端。如果服务器启动，服务就启动了，但如果服务器故障，整个服务也因此而关闭。这非常不符合可靠的协作服务的承诺。出于可靠性，我们需要运行多个服务器。

幸运的是，我们可以在一台机器上运行多个服务器。我们仅仅需要做的便是配置一个更复杂的配置文件。

为了让服务器之间可以通信，服务器间需要一些联系信息。理论上，服务器可以使用多播来发现彼此，但我们想让ZooKeeper集合支持跨多个网络而不是单个网络，这样就可以支持多个集合的情况。

为了完成这些，我们将要使用以下配置文件：

```
tickTime=2000
initLimit=10
syncLimit=5
dataDir=./data
clientPort=2181
server.1=127.0.0.1:2222:2223
server.2=127.0.0.1:3333:3334
server.3=127.0.0.1:4444:4445
```

我们主要讨论最后三行对于**server.n**项的配置信息。其余配置参数将会在第10章中进行说明。

每一个**server.n**项指定了编号为**n**的ZooKeeper服务器使用的地址和端口号。每个**server.n**项通过冒号分隔为三部分，第一部分为服务器**n**的IP地址或主机名（**hostname**），第二部分和第三部分为TCP端口号，分别用于仲裁通信和群首选举。因为我们在同一个机器上运行三个服务器进程，所以我们需要在每一项中使用不同的端口号。通常，我们在不同的服务器上运行每个服务器进程，因此每个服务器项的配置可以使用相同的端口号。

我们还需要分别设置**data**目录，我们可以在命令行中通过以下命令来操作：

```
mkdir z1
mkdir z1/data
mkdir z2
mkdir z2/data
mkdir z3
mkdir z3/data
```

当启动一个服务器时，我们需要知道启动的是哪个服务器。一个服务器通过读取data目录下一个名为myid的文件来获取服务器ID信息。可以通过以下命令来创建这些文件：

```
echo 1 > z1/data/myid
echo 2 > z2/data/myid
echo 3 > z3/data/myid
```

当服务器启动时，服务器通过配置文件中的dataDir参数来查找data目录的配置。它通过mydata获得服务器ID，之后使用配置文件中server.n对应的项来设置端口并监听。当在不同的机器上运行ZooKeeper服务器进程时，它们可以使用相同的客户端端口和相同的配置文件。但对于这个例子，在一台服务器上运行，我们需要自定义每个服务器的客户端端口。

因此，首先使用本章之前讨论的配置文件，创建z1/z1.cfg。之后通过分别改变客户端端口号为2182和2183，创建配置文件z2/z2.cfg和z3/z3.cfg。

现在可以启动服务器，让我们从z1开始：

```
$ cd z1
$ {PATH_TO_ZK}/bin/zkServer.sh start ./z1.cfg
```

服务器的日志记录为zookeeper.out。因为我们只启动了三个ZooKeeper服务器中的一个，所以整个服务还无法运行。在日志中我们将会看到以下形式的记录：

```
... [myid:1] - INFO [QuorumPeer[myid=1]/...:2181:QuorumPeer@670] - LOOKING
... [myid:1] - INFO [QuorumPeer[myid=1]/...:2181:FastLeaderElection@740] -
New election. My id = 1, proposed zxid=0x0
... [myid:1] - INFO [WorkerReceiver[myid=1]:FastLeaderElection@542] -
Notification: 1 ..., LOOKING (my state)
... [myid:1] - WARN [WorkerSender[myid=1]:QuorumCnxManager@368] - Cannot
open channel to 2 at election address /127.0.0.1:3334
    Java.net.ConnectException: Connection refused
        at java.net.PlainSocketImpl.socketConnect(Native Method)
        at java.net.PlainSocketImpl.doConnect(PlainSocketImpl.java:351)
```

这个服务器疯狂地尝试连接到其他服务器，然后失败，如果我们启动另一个服务器，我们可以构成仲裁的法定人数：

```
$ cd z2
$ {PATH_TO_ZK}/bin/zkServer.sh start ./z2.cfg
```

如果我们观察第二个服务器的日志记录zookeeper.out，我们将会看到：

```
... [myid:2] - INFO [QuorumPeer[myid=2]/...:2182:Leader@345] - LEADING
- LEADER ELECTION TOOK - 279
... [myid:2] - INFO [QuorumPeer[myid=2]/...:2182:FileTxnSnapLog@240] -
Snapshotting: 0x0 to ./data/version-2/snapshot.0
```

该日志指出服务器2已经被选举为群首。如果我们现在看看服务器1的日志，我们会看到：

```
... [myid:1] - INFO [QuorumPeer[myid=1]/...:2181:QuorumPeer@738] -  
FOLLOWING  
... [myid:1] - INFO [QuorumPeer[myid=1]/...:2181:ZooKeeperServer@162] -  
Created server ...  
... [myid:1] - INFO [QuorumPeer[myid=1]/...:2181:Follower@63] - FOLLOWING  
- LEADER ELECTION TOOK - 212
```

服务器1作为服务器2的追随者被激活。我们现在具有了符合法定仲裁（三分之二）的可用服务器。

在此刻服务开始可用。我们现在需要配置客户端来连接到服务上。连接字符串需要列出所有组成服务的服务器host: port对。对于这个例子，连接串为"127.0.0.1: 2181, 127.0.0.1: 2182, 127.0.0.1: 2183"（我们包含第三个服务器的信息，即使我们永远不启动它，因为这可以说明ZooKeeper一些有用的属性）。

我们使用zkCli.sh来访问集群：

```
$ {PATH_TO_ZK}/bin/zkCli.sh -server 127.0.0.1:2181,127.0.0.1:2182,127.0.0.1:2183
```

当连接到服务器后，我们会看到以下形式的消息：

```
[myid:] - INFO [...] - Session establishment  
complete on server localhost/127.0.0.1:2182 ...
```

注意日志消息中的端口号，在本例中的2182。如果通过Ctrl-C来停止客户端并重启多次它，我们将会看到端口号在2181和2182之间来回变化。我们也许还会注意到尝试2183端口后连接失败的消息，之后为成功连接到某一个服务器端口的消息。

注意：简单的负载均衡

客户端以随机顺序连接到连接串中的服务器。这样可以用 **ZooKeeper** 来实现一个简单的负载均衡。不过，客户端无法指定优先选择的服务器来进行连接。例如，如果有5个 **ZooKeeper** 服务器的一个集合，其中3个在美国西海岸，另外两个在美国东海岸，为了确保客户端只连接到本地服务器上，我们可以使在东海岸客户端的连接串中只出现东海岸的服务器，在西海岸客户端的连接串中只有西海岸的服务器。

这个连接尝试说明如何通过运行多个服务器来达到可靠性（当然，在生产环境中，你需要在不同的主机上进行这些操作）。对于本书大部分，包括后续几章，我们一直以独立模式的服务器进行开发，因为启动和管理多个服务器非常简单，实现这个例子也非常简单。除了连接串外，客户端不用关心 **ZooKeeper** 服务由多少个服务器组成，这也是 **ZooKeeper** 的优点之一。

2.3.4 实现一个原语：通过 **ZooKeeper** 实现锁

关于 **ZooKeeper** 的功能，一个简单的例子就是通过锁来实现临界区域。我们知道有很多形式的锁（如：读/写锁、全局锁），通过 **ZooKeeper** 来实现锁也有多种方式。这里讨论一个简单的方式来说明应用中如何使用 **ZooKeeper**，我们不再考虑其他形式的锁。

假设有一个应用由n个进程组成，这些进程尝试获取一个锁。再次强调，ZooKeeper并未直接暴露原语，因此我们使用ZooKeeper的接口来管理znode，以此来实现锁。为了获得一个锁，每个进程p尝试创建znode，名为/lock。如果进程p成功创建了znode，就表示它获得了锁并可以继续执行其临界区域的代码。不过一个潜在的问题是进程p可能崩溃，导致这个锁永远无法释放。在这种情况下，没有任何其他进程可以再次获得这个锁，整个系统可能因死锁而失灵。为了避免这种情况，我们不得不在创建这个节点时指定/lock为临时节点。

其他进程因znode存在而创建/lock失败。因此，进程监听/lock的变化，并在检测到/lock删除时再次尝试创建节点来获得锁。当收到/lock删除的通知时，如果进程p还需要继续获取锁，它就继续尝试创建/lock的步骤，如果其他进程已经创建了，就继续监听节点。

2.4 一个主-从模式例子的实现

本节中我们通过zkCli工具来实现主-从示例的一些功能。这个例子仅用于教学目的，我们不推荐使用zkCli工具来搭建系统。使用zkCli的目的仅仅是为了说明如何通过ZooKeeper来实现协作菜谱，从而撇开在实际实现中所需的大量细节。我们将在下一章中进入实现的细节。

主-从模式的模型中包括三个角色：

主节点

主节点负责监视新的从节点和任务，分配任务给可用的从节点。

从节点

从节点会通过系统注册自己，以确保主节点看到它们可以执行任务，然后开始监视新任务。

客户端

客户端创建新任务并等待系统的响应。

现在探讨这些不同的角色以及每个角色需要执行的确切步骤。

2.4.1 主节点角色

因为只有一个进程会成为主节点，所以一个进程成为ZooKeeper的主节点后必须锁定管理权。为此，进程需要创建一个临时znode，名为/master:

```
[zk: localhost:2181(CONNECTED) 0] create -e /master "master1.example.com:2223"①
```

```
Created /master
[zk: localhost:2181(CONNECTED) 1] ls /②
```

```
[master, zookeeper]
[zk: localhost:2181(CONNECTED) 2] get /master③
```

```
"master1.example.com:2223"
cZxid = 0x67
ctime = Tue Dec 11 10:06:19 CET 2012
mZxid = 0x67
mtime = Tue Dec 11 10:06:19 CET 2012
pZxid = 0x67
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x13b891d4c9e0005
dataLength = 26
numChildren = 0
[zk: localhost:2181(CONNECTED) 3]
```

①创建主节点的znode，以便获得管理权。使用-e标志来表示创建的znode为临时性的。

②列出ZooKeeper树的根。

③获取/master znode的元数据和数据。

刚刚发生了什么？首先创建一个临时znode/master。我们在znode中添加了主机信息，以便ZooKeeper外部的其他进程需要与它通信。添加主机信息并不是必需的，但这样做仅仅是为了说明我们可以在需要时添加数据。为了设置znode为临时性的，需要添加-e标志。记得，一个临时节点会在会话过期或关闭时自动被删除。

现在让我们看下我们使用两个进程来获得主节点角色的情况，尽管在任何时刻最多只能有一个活动的主节点，其他进程将成为备份主节点。假如其他进程不知道已经有一个主节点被选举出来，并尝试创建一个/master节点。让我们看看会发生什么：

```
[zk: localhost:2181(CONNECTED) 0] create -e /master "master2.example.com:2223"
Node already exists: /master
[zk: localhost:2181(CONNECTED) 1]
```

ZooKeeper告诉我们一个/master节点已经存在。这样，第二个进程就知道已经存在一个主节点。然而，一个活动的主节点可能会崩溃，备份主节点需要接替活动主节点的角色。为了检测到这些，需要在/master节点上设置一个监视点，操作如下：

```
[zk: localhost:2181(CONNECTED) 0] create -e /master "master2.example.com:2223"
Node already exists: /master
[zk: localhost:2181(CONNECTED) 1] stat /master true
```

```
cZxid = 0x67
ctime = Tue Dec 11 10:06:19 CET 2012
mZxid = 0x67
mtime = Tue Dec 11 10:06:19 CET 2012
pZxid = 0x67
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x13b891d4c9e0005
dataLength = 26
numChildren = 0
[zk: localhost:2181(CONNECTED) 2]
```

`stat`命令可以得到一个`znode`节点的属性，并允许我们在已经存在的`znode`节点上设置监视点。通过在路径后面设置参数`true`来添加监视点。当活动的主节点崩溃时，我们会观察到以下情况：

```
[zk: localhost:2181(CONNECTED) 0] create -e /master "master2.example.com:2223"
Node already exists: /master
[zk: localhost:2181(CONNECTED) 1] stat /master true
cZxid = 0x67
ctime = Tue Dec 11 10:06:19 CET 2012
mZxid = 0x67
mtime = Tue Dec 11 10:06:19 CET 2012
pZxid = 0x67
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x13b891d4c9e0005
dataLength = 26
numChildren = 0
[zk: localhost:2181(CONNECTED) 2]
WATCHER::
WatchedEvent state:SyncConnected type:NodeDeleted path:/master
[zk: localhost:2181(CONNECTED) 2] ls /
[zookeeper]
[zk: localhost:2181(CONNECTED) 3]
```

在输出的最后，我们注意到`NodeDeleted`事件。这个事件指出活动主节点的会话已经关闭或过期。同时注意，`/master`节点已经不存在了。现在备份主节点通过再次创建`/master`节点来成为活动主节点。

```
[zk: localhost:2181(CONNECTED) 0] create -e /master "master2.example.com:2223"
Node already exists: /master
```

```
[zk: localhost:2181(CONNECTED) 1] stat /master true
cZxid = 0x67
ctime = Tue Dec 11 10:06:19 CET 2012
mZxid = 0x67
mtime = Tue Dec 11 10:06:19 CET 2012
pZxid = 0x67
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x13b891d4c9e0005
dataLength = 26
numChildren = 0
[zk: localhost:2181(CONNECTED) 2]
WATCHER::
WatchedEvent state:SyncConnected type:NodeDeleted path:/master
[zk: localhost:2181(CONNECTED) 2] ls /
[zookeeper]
[zk: localhost:2181(CONNECTED) 3] create -e /master "master2.example.com:2223"
Created /master
[zk: localhost:2181(CONNECTED) 4]
```

因为备份主节点成功创建了/**master**节点，所以现在客户端开始成为活动主节点。

2.4.2 从节点、任务和分配

在我们讨论从节点和客户端所采取的步骤之前，让我们先创建三个重要的父**znode**，/**workers**、/**tasks**和/**assign**:

```
[zk: localhost:2181(CONNECTED) 0] create /workers ""
Created /workers
[zk: localhost:2181(CONNECTED) 1] create /tasks ""
Created /tasks
[zk: localhost:2181(CONNECTED) 2] create /assign ""
Created /assign
[zk: localhost:2181(CONNECTED) 3] ls /
[assign, tasks, workers, master, zookeeper]
[zk: localhost:2181(CONNECTED) 4]
```

这三个新的**znode**为持久性节点，且不包含任何数据。本例中，通过使用这些**znode**可以告诉我们哪个从节点当前有效，还告诉我们当前

有任务需要分配，并向从节点分配任务。

在真实的应用中，这些`znode`可能由主进程在分配任务前创建，也可能由一个引导程序创建，不管这些节点是如何创建的，一旦这些节点存在了，主节点就需要监视/`workers`和/`tasks`的子节点的变化情况：

```
[zk: localhost:2181(CONNECTED) 4] ls /workers true
[]
[zk: localhost:2181(CONNECTED) 5] ls /tasks true
[]
[zk: localhost:2181(CONNECTED) 6]
```

请注意，在主节点上调用`stat`命令前，我们使用可选的`true`参数调用`ls`命令。通过`true`这个参数，可以设置对应`znode`的子节点变化的监视点。

2.4.3 从节点角色

从节点首先要通知主节点，告知从节点可以执行任务。从节点通过在/`workers`子节点下创建临时性的`znode`来进行通知，并在子节点中使用主机名来标识自己：

```
[zk: localhost:2181(CONNECTED) 0] create -e /workers/worker1.example.com
                                         "worker1.example.com:2224"
Created /workers/worker1.example.com
[zk: localhost:2181(CONNECTED) 1]
```

注意，输出中，ZooKeeper确认znode已经创建。之前主节点已经监视了/workers的子节点变化情况。一旦从节点在/workers下创建了一个znode，主节点就会观察到以下通知信息：

```
WATCHER::
WatchedEvent state:SyncConnected type:NodeChildrenChanged path:/workers
```

下一步，从节点需要创建一个父znode/assign/worker1.example.com来接收任务分配，并通过第二个参数为true的ls命令来监视这个节点的变化，以便等待新的任务。

```
[zk: localhost:2181(CONNECTED) 0] create -e /workers/worker1.example.com
                                         "worker1.example.com:2224"
Created /workers/worker1.example.com
[zk: localhost:2181(CONNECTED) 1] create /assign/worker1.example.com ""
Created /assign/worker1.example.com
[zk: localhost:2181(CONNECTED) 2] ls /assign/worker1.example.com true
[]
[zk: localhost:2181(CONNECTED) 3]
```

从节点现在已经准备就绪，可以接收任务分配。之后，我们通过讨论客户端角色来看一下任务分配的问题。

2.4.4 客户端角色

客户端向系统中添加任务。在本示例中具体任务是什么并不重要，我们假设客户端请求主从系统来运行cmd命令。为了向系统添加一个任务，客户端执行以下操作：

```
[zk: localhost:2181(CONNECTED) 0] create -s /tasks/task- "cmd"
Created /tasks/task-0000000000
```

我们需要按照任务添加的顺序来添加**znode**，其本质上为一个队列。客户端现在必须等待任务执行完毕。执行任务的从节点将任务执行完毕后，会创建一个**znode**来表示任务状态。客户端通过查看任务状态的**znode**是否创建来确定任务是否执行完毕，因此客户端需要监视状态**znode**的创建事件：

```
[zk: localhost:2181(CONNECTED) 1] ls /tasks/task-0000000000 true
[]
[zk: localhost:2181(CONNECTED) 2]
```

执行任务的从节点会在/tasks/task-0000000000节点下创建状态**znode**节点，所以我们需要用ls命令来监视/tasks/task-0000000000的子节点。

一旦创建任务的**znode**，主节点会观察到以下事件：

```
[zk: localhost:2181(CONNECTED) 6]
WATCHER::
WatchedEvent state:SyncConnected type:NodeChildrenChanged path:/tasks
```

主节点之后会检查这个新的任务，获取可用的从节点列表，之后分配这个任务给worker1.example.com：

```
[zk: 6] ls /tasks
[task-0000000000]
[zk: 7] ls /workers
[worker1.example.com]
[zk: 8] create /assign/worker1.example.com/task-0000000000 ""
```

```
Created /assign/worker1.example.com/task-0000000000  
[zk: 9]
```

从节点接收到新任务分配的通知:

```
[zk: localhost:2181(CONNECTED) 3]  
WATCHER::  
WatchedEvent state:SyncConnected type:NodeChildrenChanged  
path:/assign/worker1.example.com
```

从节点之后便开始检查新任务，并确认该任务是否分配给自己:

```
WATCHER::  
WatchedEvent state:SyncConnected type:NodeChildrenChanged  
path:/assign/worker1.example.com  
[zk: localhost:2181(CONNECTED) 3] ls /assign/worker1.example.com  
[task-0000000000]  
[zk: localhost:2181(CONNECTED) 4]
```

一旦从节点完成任务的执行，它就会在/tasks中添加一个状态

znode:

```
[zk: localhost:2181(CONNECTED) 4] create /tasks/task-0000000000/status "done"  
Created /tasks/task-0000000000/status  
[zk: localhost:2181(CONNECTED) 5]
```

之后，客户端接收到通知，并检查执行结果:

```
WATCHER::  
WatchedEvent state:SyncConnected type:NodeChildrenChanged  
path:/tasks/task-0000000000  
[zk: localhost:2181(CONNECTED) 2] get /tasks/task-0000000000  
"cmd"  
cZxid = 0x7c  
ctime = Tue Dec 11 10:30:18 CET 2012  
mZxid = 0x7c  
mtime = Tue Dec 11 10:30:18 CET 2012  
pZxid = 0x7e  
cversion = 1
```

```
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 5
numChildren = 1
[zk: localhost:2181(CONNECTED) 3] get /tasks/task-0000000000/status
"done"
cZxid = 0x7e
ctime = Tue Dec 11 10:42:41 CET 2012
mZxid = 0x7e
mtime = Tue Dec 11 10:42:41 CET 2012
pZxid = 0x7e
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 8
numChildren = 0
[zk: localhost:2181(CONNECTED) 4]
```

客户端检查状态znode的信息，并确认任务的执行结果。本例中，我们看到任务成功执行，其状态为“done”。当然任务也可能非常复杂，甚至涉及另一个分布式系统。最终不管是什么样的任务，执行任务的机制与通过ZooKeeper来传递结果，本质上都是一样的。

2.5 小结

本章中，我们了解了许多基础的ZooKeeper概念，我们看到了ZooKeeper通过其API提供的基本功能，还探讨了其架构中的一些重要概念，如通过仲裁理论进行复制。此时此刻，最重要的并不是了解ZooKeeper的复制协议是如何工作的，最重要的是明白仲裁理论的概念，因为你在部署ZooKeeper时需要指定服务器的数量。讨论的另一个重要概念便是会话。会话的语义对ZooKeeper的保障非常关键，因为它们常常会涉及会话。

为了对如何使用ZooKeeper提供初步的了解，我们使用zkCli工具来访问ZooKeeper服务器，并执行请求。我们展示了使用该工具在主-从模式例子中的主要操作。当实现一个真实的ZooKeeper应用时，你不应该使用这个工具；这个工具更多地用于调试和监控目的。你需要使用ZooKeeper提供的某一语言套件。在下一章中，我们将使用Java来实现我们的例子。

第二部分 使用ZooKeeper进行开发

本书该部分可供程序员阅读，来学习如何使用ZooKeeper来协作分布式程序的开发技能和正确方法。ZooKeeper提供了Java语言和C语言的API套件，这两个套件拥有相同的基础结构和特性。Java套件最流行且简单易用，因此相关例子中均使用该套件。第7章会介绍C语言套件。主-从模式例子的源代码，可以从GitHub仓库（<https://github.com/fpi/zookeeper-book-example>）中获得。

第3章 开始使用ZooKeeper的API

在之前的章节中，我们使用zkCli工具介绍了ZooKeeper的基本操作。从本章开始，我们将会看到在应用中如何通过API来进行操作。首先介绍一下如何使用ZooKeeper的API进行开发，展示如何创建会话，实现监视点（**watcher**）。我们还是从主-从模式例子开始进行编码。

3.1 设置ZooKeeper的CLASSPATH

我们需要设置正确的classpath，以便运行或编译ZooKeeper的Java代码。除了ZooKeeper的JAR包外，ZooKeeper使用了大量的第三方库。为了简化输入和方便阅读，我们使用环境变量CLASSPATH来表示所有必需的库。ZooKeeper发行包中bin目录下的zkEnv.sh脚本会为我们设置该环境变量。我们需要使用以下方式编码：

```
ZOOBINDIR="<path_to_distro>/bin"  
. "$ZOOBINDIR"/zkEnv.sh
```

（在Windows上，使用call命令调用，而不是使用zkEnv.cmd脚本。）

一旦运行这个脚本，环境变量CLASSPATH就会正确设置。我们在编译和运行Java程序时用到它。

3.2 建立ZooKeeper会话

ZooKeeper的API围绕ZooKeeper的句柄（handle）而构建，每个API调用都需要传递这个句柄。这个句柄代表与ZooKeeper之间的一个会话。在图3-1中，与ZooKeeper服务器已经建立的一个会话如果断开，这个会话就会迁移到另一台ZooKeeper服务器上。只要会话还存活着，这个句柄就仍然有效，ZooKeeper客户端库会持续保持这个活跃连接，以保证与ZooKeeper服务器之间的会话存活。如果句柄关闭，ZooKeeper客户端库会告知ZooKeeper服务器终止这个会话。如果ZooKeeper发现客户端已经死掉，就会使这个会话无效。如果客户端之后尝试重新连接到ZooKeeper服务器，使用之前无效会话对应的那个句柄进行连接，那么ZooKeeper服务器会通知客户端库，这个会话已失效，使用这个句柄进行的任何操作都会返回错误。

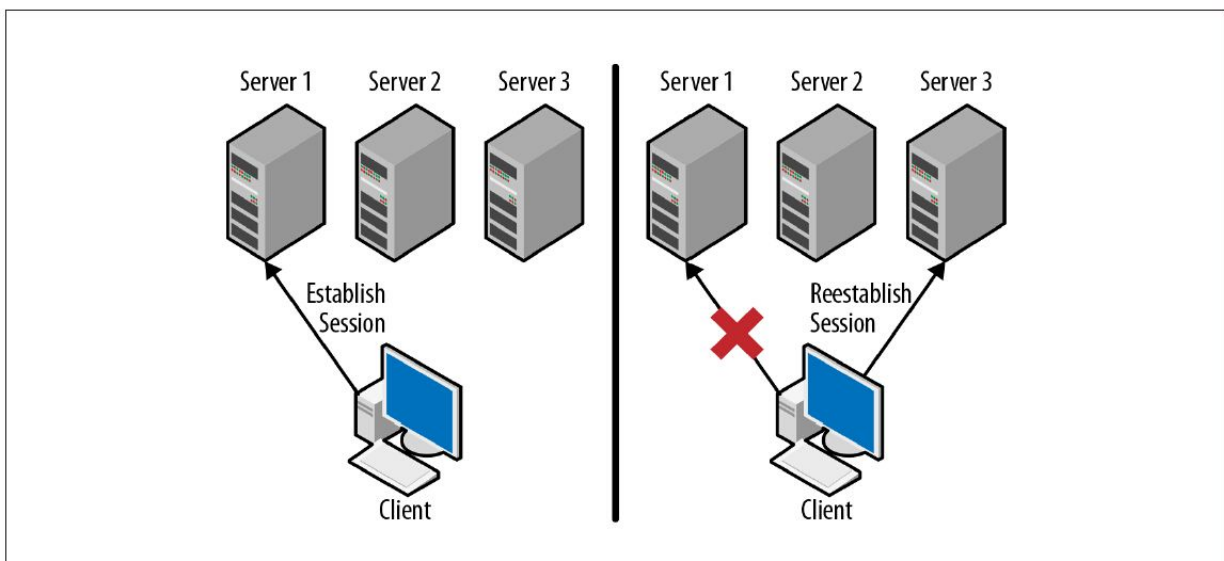


图3-1：会话在两个服务器之间发生迁移

创建ZooKeeper句柄的构造函数如下所示：

```
ZooKeeper(  
    String connectString,  
    int sessionTimeout,  
    Watcher watcher)
```

其中：

connectString

包含主机名和ZooKeeper服务器的端口。我们之前通过zkCli连接ZooKeeper服务时，已经列出过这些服务器。

sessionTimeout

以毫秒为单位，表示ZooKeeper等待客户端通信的最长时间，之后会声明会话已死亡。目前我们使用15000，即15秒。这就是说如果ZooKeeper与客户端有15秒的时间无法进行通信，ZooKeeper就会终止客户端的会话。需要注意，这个值比较高，但对于我们后续的实验会非常有用。ZooKeeper会话一般设置超时时间为5~10秒。

watcher

用于接收会话事件的一个对象，这个对象需要我们自己创建。因为Watcher定义为接口，所以我们需要自己实现一个类，然后初始化这

个类的实例并传入ZooKeeper的构造函数中。客户端使用Watcher接口来监控与ZooKeeper之间会话的健康情况。与ZooKeeper服务器之间建立或失去连接时就会产生事件。它们同样还能用于监控ZooKeeper数据的变化。最终，如果与ZooKeeper的会话过期，也会通过Watcher接口传递事件来通知客户端的应用。

3.2.1 实现一个Watcher

为了从ZooKeeper接收通知，我们需要实现监视点。首先让我们进一步了解Watcher接口，该接口的定义如下：

```
public interface Watcher {  
    void process(WatchedEvent event);  
}
```

这个接口没有多少内容，我们不得不自己实现，但现在我们只是简单地输出事件。所以，让我们从一个名为Master的类开始实现示例：

```
import org.apache.zookeeper.ZooKeeper;  
import org.apache.zookeeper.Watcher;  
public class Master implements Watcher {  
    ZooKeeper zk;  
    String hostPort;  
    Master(String hostPort) {  
        this.hostPort = hostPort;①  
    }  
    void startZK() {  
        zk = new ZooKeeper(hostPort, 15000, this);②  
    }  
}
```

```

    }
    public void process(WatchedEvent e) {
        System.out.println(e);③

    }

    public static void main(String args[])
        throws Exception {
        Master m = new Master(args[0]);
        m.startZK();
        // wait for a bit
        Thread.sleep(60000);④
    }
}

```

①在构造函数中，我们并未实例化ZooKeeper对象，而是先保存hostPort留着后面使用。Java最佳实践告诉我们，一个对象的构造函数没有完成前不要调用这个对象的其他方法。因为这个对象实现了Watcher，并且当我们实例化ZooKeeper对象时，其Watcher的回调函数就会被调用，所以我们需要Master的构造函数返回后再调用ZooKeeper的构造函数。

②使用Master对象来构造ZooKeeper对象，以便添加Watcher的回调函数。

③这个简单的示例没有提供复杂的事件处理逻辑，而只是将我们收到的事件进行简单的输出。

④我们连接到ZooKeeper后，后台就会有一个线程来维护这个ZooKeeper会话。该线程为守护线程，也就是说线程即使处于活跃状态，程序也可以退出。因此我们在程序退出前休眠一段时间，以便我们可以看到事件的发生。

通过以下方式就可以编译这个简单的例子：

```
$ javac -cp $CLASSPATH Master.java
```

当我们编译完Master.java这个文件后，运行它并查看结果：

```
$ java -cp $CLASSPATH Master 127.0.0.1:2181
... - INFO [...] - Client environment:zookeeper.version=3.4.5-1392090, ...①

...
... - INFO [...] - Initiating client connection,
connectString=127.0.0.1:2181 ...②

... - INFO [...] - Opening socket connection to server
localhost/127.0.0.1:2181. ...
... - INFO [...] - Socket connection established to localhost/127.0.0.1:2181,
initiating session
... - INFO [...] - Session establishment complete on server
localhost/127.0.0.1:2181, ...③

WatchedEvent state:SyncConnected type:None path:null④
```

ZooKeeper客户端API产生很多日志消息，使用户可以了解发生了什么。日志非常详细，可以通过配置文件来禁用这么详细的日志，不过在开发时，这些消息非常有用，甚至在正式部署后，发生了某些我们不希望发生的事情，这些日志消息也是非常有价值的。

①前面几行日志消息描述了ZooKeeper客户端的实现和环境。

②当客户端初始化一个到ZooKeeper服务器的连接时，无论是最初的连接还是随后的重连接，都会产生这些日志消息。

③这个消息展示了连接建立之后，该连接的信息，其中包括客户端所连接的主机和端口信息，以及这个会话与服务器协商的超时时间。如果服务器发现请求的会话超时时间太短或太长，服务器会调整会话超时时间。

④最后这行并不是ZooKeeper库所输出，而是我们实现的Watcher.process（WatchedEvent e）函数中输出的WatchEvent对象。

这个例子中，假设运行时所有必需的库均在lib子目录下，同时假设log4j.conf文件在conf子目录中。你可以在你所使用的ZooKeeper发行包中找到这两个目录。如果你看到以下信息：

```
log4j:WARN No appenders could be found for logger
(org.apache.zookeeper.ZooKeeper).
log4j:WARN Please initialize the log4j system properly.
```

表示你还没有将log4j.conf放到classpath下。

3.2.2 运行Watcher的示例

如果我们不启动ZooKeeper服务就启动主节点，这样会发生什么呢？我们可以试一下。停止服务，然后运行Master，看到了什么？在之前输出中的最后一行，即WatchedEvent的数据，现在并没有出现。因为ZooKeeper库无法连接ZooKeeper服务器，所以我们看不到这些信息了。

现在我们启动服务器，然后运行Master，之后停止服务器并保持Master继续运行。你会看到在SyncConnected事件之后发生了Disconnected事件。

当开发者看到Disconnected事件时，有些人认为需要创建一个新的ZooKeeper句柄来重新连接服务。不要这么做！当你启动服务器，然后启动Master，再重启服务器时看一下发生了什么。你看到SyncConnected事件之后为Disconnected事件，然后又是一个SyncConnected事件。ZooKeeper客户端库负责为你重新连接服务。当不幸遇到网络中断或服务器故障时，ZooKeeper可以处理这些故障问题。

我们需要知道ZooKeeper本身也可能发生这些故障问题。一个ZooKeeper服务器也许会故障或失去网络连接，类似我们停止主节点后

所模拟的场景。如果ZooKeeper服务至少由三台服务器组成，那么一个服务器的故障并不会导致服务中断。而客户端也会很快收到Disconnected事件，之后便为SyncConnected事件。

注意：ZooKeeper管理连接

请不要自己试着去管理ZooKeeper客户端连接。ZooKeeper客户端库会监控与服务之间的连接，客户端库不仅告诉我们连接发生问题，还会主动尝试重新建立通信。一般客户端开发库会很快重建会话，以便最小化应用的影响。所以不要关闭会话后再启动一个新的会话，这样会增加系统负载，并导致更长时间的中断。

客户端就像除了休眠外什么都没做一样，而我们通过发生的事件可以看到后台到底发生了什么。我们还可以看看ZooKeeper服务端都发生了什么。ZooKeeper有两种管理接口：JMX和四字母组成的命令。第10章会深入讨论这些接口，现在我们通过stat和dump这两个四字母命令来看看服务器上发生了什么。

要使用这些命令，需要先通过telnet连接到客户端端口2181，然后输入这些命令（在命令后输入Enter键）。例如，如果启动Master这个程序后，使用stat命令，我们会看到以下输出信息：

```
$ telnet 127.0.0.1 2181
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
stat
```

```
ZooKeeper version: 3.4.5-1392090, built on 09/30/2012 17:52 GMT
Clients:
  /127.0.0.1:39470[1](queued=0,recved=3,sent=3)
  /127.0.0.1:39471[0](queued=0,recved=1,sent=0)
Latency min/avg/max: 0/5/48
Received: 34
Sent: 33
Connections: 2
Outstanding: 0
Zxid: 0x17
Mode: standalone
Node count: 4
Connection closed by foreign host.
```

我们从输出信息看到有两个客户端连接到ZooKeeper服务器。一个是Master程序，另一个为Telnet连接。

如果我们启动Master程序后，使用dump命令，我们会看到以下输出信息：

```
$ telnet 127.0.0.1 2181
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
dump
SessionTracker dump:
Session Sets (3):
0 expire at Wed Nov 28 20:34:00 PST 2012:
0 expire at Wed Nov 28 20:34:02 PST 2012:
1 expire at Wed Nov 28 20:34:04 PST 2012:
    0x13b4a4d22070006
ephemeral nodes dump:
Sessions with Ephemerals (0):
Connection closed by foreign host.
```

我们从输出信息中看到有一个活动的会话，这个会话属于Master程序。我们还能看到这个会话还有多长时间会过期。会话超时的过期时间取决于我们创建ZooKeeper对象时所指定的值。

让我结束**Master**程序，再次使用**dump**命令来看一下活动的会话信息。你会注意到会话过一段时间后才消失。这是因为直到会话超时时间过了以后，服务器才会结束这个会话。当然，客户端会不断延续与**ZooKeeper**服务器的活动连接的到期时间。

当**Master**结束时，最好的方式是使会话立即消失。这可以通过**ZooKeeper.close ()**方法来结束。一旦调用**close**方法后，**ZooKeeper**对象实例所表示的会话就会被销毁。

让我们在示例程序中加入**close**调用：

```
void stopZK() throws Exception { zk.close(); }
public static void main(String args[]) throws Exception {
    Master m = new Master(args[0]);
    m.startZK();
    // wait for a bit
    Thread.sleep(60000);
    m.stopZK();
}
```

现在我们可以再次运行**Master**程序，运行**dump**命令来看一下会话是否还存活。因为**Master**程序中显示关闭了会话，所以**ZooKeeper**关闭会话前就不需要等待会话超时了。

3.3 获取管理权

现在我们有了会话，我们的**Master**程序需要获得管理权，虽然现在只有一个主节点，但我们还是要小心仔细。我们需要运行多个进程，以便在活动主节点发生故障后，可以有进程接替主节点。

为了确保同一时间只有一个主节点进程出于活动状态，我们使用**ZooKeeper**来实现简单的群首选举算法（在2.4.1节中所描述的）。这个算法中，所有潜在的主节点进程尝试创建/**master**节点，但只有一个成功，这个成功的进程成为主节点。

常量**ZooDefs.Ids.OPEN_ACL_UNSAFE**为所有人提供了所有权限（正如其名所显示的，这个**ACL**策略在不可信的环境下使用是非常不安全的）。

ZooKeeper通过插件式的认证方法提供了每个节点的**ACL**策略功能，因此，如果我们需要，就可以限制某个用户对某个**znode**节点的哪些权限，但对于这个简单的例子，我们继续使用**OPEN_ACL_UNSAFE**策略。当然，我们希望在主节点死掉后/**master**节点会消失。正如我们在2.1.2节中所提到的持久性和临时性**znode**节点，我们可以使用**ZooKeeper**的临时性**znode**节点来达到我们的目的。我们将定义一个

EPHEMERAL的znode节点，当创建它的会话关闭或无效时，ZooKeeper会自动检测到，并删除这个节点。

因此，我们将会在我们的程序中添加以下代码：

```
String serverId = Integer.toHexString(random.nextInt());
void runForMaster() {
    zk.create("/master", ①

    serverId.getBytes(), ②

    OPEN_ACL_UNSAFE, ③

    CreateMode.EPHEMERAL); ④

}
```

①我们试着创建znode节点/master。如果这个znode节点存在，create就会失败。同时我们想在/master节点的数据字段保存对应这个服务器的唯一ID。

②数据字段只能存储字节数组类型的数据，所以我们将int型转换为一个字节数组。

③如之前所提到的，我们使用开放的ACL策略。

④我们创建的节点类型为EPHEMERAL。

然而，我们这样做还不够，`create`方法会抛出两种异常：

`KeeperException`和`InterruptedException`。我们需要确保我们处理了这两种异常，特别是`ConnectionLossException`（`KeeperException`异常的子类）和`InterruptedException`。对于其他异常，我们可以忽略并继续执行，但对于这两种异常，`create`方法可能已经成功了，所以如果我们作为主节点就需要捕获并处理它们。

`ConnectionLossException`异常发生于客户端与ZooKeeper服务端失去连接时。一般常常由于网络原因导致，如网络分区或ZooKeeper服务器故障。当这个异常发生时，客户端并不知道是在ZooKeeper服务器处理前丢失了请求消息，还是在处理后客户端未收到响应消息。如我们之前所描述的，ZooKeeper的客户端库将会为后续请求重新建立连接，但进程必须知道一个未决请求是否已经处理了还是需要再次发送请求。

`InterruptedException`异常源于客户端线程调用了`Thread.interrupt`，通常这是因为应用程序部分关闭，但还在被其他相关应用的方法使用。从字面来看这个异常，进程会中断本地客户端的请求处理的过程，并使该请求处于未知状态。

这两种请求都会导致正常请求处理过程的中断，开发者不能假设处理过程中的请求的状态。当我们处理这些异常时，开发者在处理前必须知道系统的状态。如果发生群首选举，在我们没有确认情况之前，我们不希望确定主节点。如果create执行成功了，活动主节点死掉以前，没有任何进程能够成为主节点，如果活动主节点还不知道自己已经获得了管理权，不会有任何进程成为主节点进程。

当处理ConnectionLossException异常时，我们需要找出那个进程创建的/master节点，如果进程是自己，就开始成为群首角色。我们通过getData方法来处理：

```
byte[] getData(  
    String path,  
    bool watch,  
    Stat stat)
```

其中：

[path](#)

类似其他ZooKeeper方法一样，第一个参数为我们想要获取数据的znode节点路径。

[watch](#)

表示我们是否想要监听后续的数据变更。如果设置为`true`，我们就可以通过我们创建`ZooKeeper`句柄时所设置的`Watcher`对象得到事件，同时另一个版本的方法提供了以`Watcher`对象为入参，通过这个传入的对象来接收变更的事件。我们在后续章节再讨论如何监视变更情况，现在我们设置这个参数为`false`，因为我们现在只想知道当前的数据是什么。

stat

最后一个参数类型`Stat`结构，`getData`方法会填充`znode`节点的元数据信息。

返回值

方法返回成功（没有抛出异常），就会得到`znode`节点数据的字节数组。

让我们按以下代码段来修改代码，在`runForMaster`方法中引入异常处理：

```
String serverId = Integer.toString(Random.nextLong());
boolean isLeader = false;
// returns true if there is a master
boolean checkMaster() {
    while (true) {
        try {
            Stat stat = new Stat();
            byte data[] = zk.getData("/master", false, stat);①
```

```

        isLeader = new String(data).equals(serverId));②

        return true;
    } catch (NoNodeException e) {
        // no master, so try create again
        return false;
    } catch (ConnectionLossException e) {
    }
}
}
void runForMaster() throws InterruptedException {③

    while (true) {
        try {④

            zk.create("/master", serverId.getBytes(),
                OPEN_ACL_UNSAFE, CreateMode.EPHEMERAL);⑤

            isLeader = true;
            break;
        } catch (NodeExistsException e) {
            isLeader = false;
            break;
        } catch (ConnectionLossException e) {⑥

        }
        if (checkMaster()) break;⑦

    }
}

```

①通过获取/master节点的数据来检查活动主节点。

②该行展示了为什么我们需要使用在创建/**master**节点时保存的数据：如果/**master**存在，我们使用/**master**中的数据来确定谁是群首。如果一个进程捕获到**ConnectionLossException**，这个进程可能就是主节点，因**create**操作实际上已经处理完，但响应消息却丢失了。

③我们将**InterruptedException**异常简单地传递给调用者。

④我们将**zk.create**方法包在**try**块之中，以便我们捕获并处理**ConnectionLossException**异常。

⑤这里为**create**请求，如果成功执行将会成为主节点。

⑥处理**ConnectionLossException**异常的**catch**块的代码为空，因为我们并不想中止函数，这样就可以使处理过程继续向下执行。

⑦检查活动主节点是否存在，如果不存在就重试。

在这个例子中，我们简单地传递**InterruptedException**给调用者，即向上传递异常。不过，在**Java**中没有明确的指导方针告诉我们如何处理线程中断，甚至没有告诉我们这个中断代表什么。有些时候，中断用于通知线程现在要退出了，需要进行清理操作，另外的情况，中断用于获得一个线程的控制权，应用的执行还将继续。

InterruptedException异常的处理依赖于程序的上下文环境，如果向上抛出**InterruptedException**异常，最终关闭**zk**句柄，我们可以抛出异常

到调用栈顶，当句柄关闭时就可以清理所有一切。如果zk句柄未关闭，在重新抛出异常前，我们需要弄清楚自己是不是主节点，或者继续异步执行后续操作。后者情况非常棘手，需要我们仔细设计并妥善处理。

现在，我们看一下Master的main主函数：

```
public static void main(String args[]) throws Exception {
    Master m = new Master(args[0]);
    m.startZK();
    m.runForMaster();①

    if (isLeader) {
        System.out.println("I'm the leader");②

        // wait for a bit
        Thread.sleep(60000);
    } else {
        System.out.println("Someone else is the leader");
    }
    m.stopZK();
}
```

①调用我们之前实现的runForMaster函数，当前进程成为主节点或另一进程成为主节点后返回。

②当我们开发主节点的应用逻辑时，我们在此处开始执行这些逻辑，现在我们仅仅输出我们成为主节点的信息，然后等待60秒后退出main函数。

因为我们并没有直接处理`InterruptedException`异常，如果发生该异常，我们的进程将会简单地退出，主节点也不会再在退出前进行其他操作。在下一章，主节点开始管理系统队列中的任务，但现在我们先继续完善其他组件。

3.3.1 异步获取管理权

`ZooKeeper`中，所有同步调用方法都有对应的异步调用方法。通过异步调用，我们可以在单线程中同时进行多个调用，同时也可以简化我们的实现方式。让我们回顾管理权的例子，修改为异步调用的方式。

以下为`create`方法的异步调用版本：

```
void create(String path,
            byte[] data,
            List<ACL> acl,
            CreateMode createMode,
            AsyncCallback.StringCallback cb, ①

            Object ctx) ②
```

`create`方法的异步方法与同步方法非常相似，仅仅多了两个参数：

①提供回调方法的对象。

②用户指定上下文信息（回调方法调用是传入的对象实例）。

该方法调用后通常在**create**请求发送到服务端之前就会立即返回。回调对象通过传入的上下文参数来获取数据，当从服务器接收到**create**请求的结果时，上下文参数就会通过回调对象提供给应用程序。

注意，该**create**方法不会抛出异常，我们可以简化处理，因为调用返回前并不会等待**create**命令完成，所以我们无需关心**InterruptedException**异常；同时因请求的所有错误信息通过回调对象会第一个返回，所以我们也无需关心**KeeperException**异常。

回调对象实现只有一个方法的**StringCallback**接口：

```
void processResult(int rc, String path, Object ctx, String name)
```

异步方法调用会简单化队列对**ZooKeeper**服务器的请求，并在另一个线程中传输请求。当接收到响应信息，这些请求就会在一个专用回调线程中被处理。为了保持顺序，只会有一个单独的线程按照接收顺序处理响应包。

processResult各个参数的含义如下：

rc

返回调用的结构，返回OK或与KeeperException异常对应的编码值。

`path`

我们传给create的path参数值。

`ctx`

我们传给create的上下文参数。

`name`

创建的znode节点名称。

目前，调用成功后，`path`和`name`的值一样，但是，如果采用CreateMode.SEQUENTIAL模式，这两个参数值就不会相等。

注意：回调函数处理

因为只有一个单独的线程处理所有回调调用，如果回调函数阻塞，所有后续回调调用都会被阻塞，也就是说，一般不要在回调函数中集中操作或阻塞操作。有时，在回调函数中调用同步方法是合法的，但一般还是避免这样做，以便后续回调调用可以快速被处理。

让我们继续完成我们的主节点的功能，我们创建了
masterCreateCallback对象，用于接收create命令的结果：

```
static boolean isLeader;
static StringCallback masterCreateCallback = new StringCallback() {
    void processResult(int rc, String path, Object ctx, String name) {
        switch(Code.get(rc)) {①

            case CONNECTIONLOSS:②

                checkMaster();
                return;
            case OK:③

                isLeader = true;
                break;
            default:④

                isLeader = false;
        }
        System.out.println("I'm " + (isLeader ? "" : "not ") +
                           "the leader");
    }
};
void runForMaster() {
    zk.create("/master", serverId.getBytes(), OPEN_ACL_UNSAFE,
              CreateMode.EPHEMERAL, masterCreateCallback, null);⑤
}
```

①我们从rc参数中获得create请求的结果，并将其转换为Code枚举类型。rc如果不为0，则对应KeeperException异常。

②如果因连接丢失导致create请求失败，我们会得到CONNECTIONLOSS编码的结果，而不是ConnectionLossException异常。当连接丢失时，我们需要检查系统当前的状态，并判断我们需要如何恢复，我们将会在我们后面实现的checkMaster方法中进行处理。

③我们现在成为群首，我们先简单地赋值isLeader为true。

④其他情况，我们并未成为群首。

⑤在runForMaster方法中，我们将masterCreateCallback传给create方法，传入null作为上下文对象参数，因为在runForMaster方法中，我们现在不需要向masterCreateCallback.processResult方法传入任何信息。

我们现在需要实现checkMaster方法，这个方法与之前的同步情况不太一样，我们通过回调方法实现处理逻辑，因此在checkMaster函数中不会看到一系列的事件，而只有getData方法。getData调用完成后，后续处理将会>DataCallback对象中继续：

```
DataCallback masterCheckCallback = new DataCallback() {
    void processResult(int rc, String path, Object ctx, byte[] data,
        Stat stat) {
        switch(Code.get(rc)) {
            case CONNECTIONLOSS:
                checkMaster();
        }
    }
}
```

```
        return;
    case NONODE:
        runForMaster();
        return;
    }
}
}
void checkMaster() {
    zk.getData("/master", false, masterCheckCallback, null);
}
```

同步方法和异步方法的处理逻辑是一样的，只是异步方法中，我们没有使用`while`循环，而是通过异步操作在回调函数中进行错误处理。

此时，同步的版本看起来比异步版本实现起来更简单，但在下一章我们会看到，应用程序常常由异步变化通知所驱动，因此最终以异步方式构建系统，反而使代码更简单。同时，异步调用不会阻塞应用程序，这样其他事务可以继续进行，甚至是提交新的ZooKeeper操作。

3.3.2 设置元数据

我们将使用异步API方法来设置元数据路径。我们的主从模型设计依赖三个目录：`/tasks`、`/assign`和`/workers`，我们可以在系统启动前通过某些系统配置来创建所有目录，或者通过在主节点程序每次启动时都创建这些目录。以下代码段会创建这些路径，例子中除了连接丢失错误的处理外没有其他错误处理：

```
public void bootstrap() {
    createParent("/workers", new byte[0]);①
```

```

        createParent("/assign", new byte[0]);
        createParent("/tasks", new byte[0]);
        createParent("/status", new byte[0]);
    }
    void createParent(String path, byte[] data) {
        zk.create(path,
            data,
            Ids.OPEN_ACL_UNSAFE,
            CreateMode.PERSISTENT,
            createParentCallback,
            data);②
    }
    StringCallback createParentCallback = new StringCallback() {
        public void processResult(int rc, String path, Object ctx, String name) {
            switch (Code.get(rc)) {
                case CONNECTIONLOSS:
                    createParent(path, (byte[]) ctx);③

                    break;
                case OK:
                    LOG.info("Parent created");
                    break;
                case NODEEXISTS:
                    LOG.warn("Parent already registered: " + path);
                    break;
                default:
                    LOG.error("Something went wrong: ",
                        KeeperException.create(Code.get(rc), path));
            }
        }
    };
};

```

①我们没有数据存入这些**znode**节点，所以只传入空的字节数组。

②因为如此，我们不用关心去跟踪每个**znode**节点对应的数据，但是往往每个路径都具有独特的数据，所以我们通过回调上下文参数对**create**操作进行跟踪数据。在**create**函数的第二个和第四个参数均传入

的data对象，也许看起来有些奇怪，但第二个参数传入的data表示要保存到znode节点的数据，而第四个参数传入的data，我们可以在createParentCallback回调函数中继续使用。

③如果回调函数中得到CONNECTIONLOSS返回码，我们通过调用createPath方法来对create操作进行重试，然而调用createPath我们需要知道之前的create调用中的data参数，因此我们通过create的第四个参数传入data，就可以将数据通过ctx对象传给回调函数。因为上下文对象与回调对象不同，我们可以使所有create操作使用同一个回调对象。

从本例中，你会注意到znode节点与文件（一个包含数据的znode节点）和目录（含有子节点的znode节点）没有什么区别，每个znode节点可以具备以上两个特点。

3.4 注册从节点

现在我们已经有了主节点，我们需要配置从节点，以便主节点可以发号施令。根据我们的设计，每个从节点会在/workers下创建一个临时性的znode节点，很简单，我们通过以下代码就可以实现。我们将使用znode节点中的数据，来指示从节点的状态：

```
import java.util.*;
import org.apache.zookeeper.AsyncCallback.DataCallback;
import org.apache.zookeeper.AsyncCallback.StringCallback;
import org.apache.zookeeper.AsyncCallback.VoidCallback;
import org.apache.zookeeper.*;
import org.apache.zookeeper.ZooDefs.Ids;
import org.apache.zookeeper.AsyncCallback.ChildrenCallback;
import org.apache.zookeeper.KeeperException.Code;
import org.apache.zookeeper.data.Stat;
import org.slf4j.*;

public class Worker implements Watcher {
    private static final Logger LOG = LoggerFactory.getLogger(Worker.class);
    ZooKeeper zk;
    String hostPort;
    String serverId = Integer.toHexString(random.nextInt());
    Worker(String hostPort) {
        this.hostPort = hostPort;
    }
    void startZK() throws IOException {
        zk = new ZooKeeper(hostPort, 15000, this);
    }
    public void process(WatchedEvent e) {
        LOG.info(e.toString() + ", " + hostPort);
    }
    void register() {
        zk.create("/workers/worker-" + serverId,
            "Idle".getBytes(), ①

            Ids.OPEN_ACL_UNSAFE,
            CreateMode.EPHEMERAL, ②
```

```

        createWorkerCallback, null);
    }
    StringCallback createWorkerCallback = new StringCallback() {
        public void processResult(int rc, String path, Object ctx,
                                String name) {
            switch (Code.get(rc)) {
                case CONNECTIONLOSS:
                    register();③

                    break;
                case OK:
                    LOG.info("Registered successfully: " + serverId);
                    break;
                case NODEEXISTS:
                    LOG.warn("Already registered: " + serverId);
                    break;
                default:
                    LOG.error("Something went wrong: "
                        + KeeperException.create(Code.get(rc), path));
            }
        }
    };
    public static void main(String args[]) throws Exception {
        Worker w = new Worker(args[0]);
        w.startZK();
        w.register();
        Thread.sleep(30000);
    }
}

```

①我们将从节点的状态信息存入代表从节点的**znode**节点中。

②如果进程死掉，我们希望代表从节点的**znode**节点得到清理，所以我们使用了**EPHEMERAL**标志，这意味着，我们简单地关注/workers就可以得到有效从节点的列表。

③因为这个进程是唯一创建表示该进程的临时性**znode**节点的进程，如果创建节点时连接丢失，进程会简单地重试创建过程。

正如我们之前所看到的，因为我们注册了一个临时性节点，如果从节点死掉，表示这个从节点znode节点也会消失，所以这是我们在从节点组成管理上所有需要做的事情。

我们将从节点状态信息存入了代表从节点的znode节点，这样我们就可以通过查询ZooKeeper来获得从节点的状态。当前，我们只有初始化和空闲状态，但是，一旦从节点开始处理某些事情，我们还需要设置其他状态信息。

以下为setStatus的实现，这个方法与之前我们看到的方法有些不同，我们希望异步方式来设置状态，以便不会延迟常规流程的操作：

```
StatCallback statusUpdateCallback = new StatCallback() {
    public void processResult(int rc, String path, Object ctx, Stat stat) {
        switch(Code.get(rc)) {
            case CONNECTIONLOSS:
                setStatus((String)ctx);①
        }
    }
};

return;
}

};
synchronized private void setStatus(String status) {
    if (status == this.status) {②

        zk.setData("/workers/" + name, status.getBytes(), -1,
            statusUpdateCallback, status);③
    }
}
```



```
public void setStatus(String status) {  
    this.status = status;④  
  
    updateStatus(status);⑤  
  
}
```

④我们将状态信息保存到本地变量中，万一更新失败，我们需要重试。

⑤我们并未在setStatus进行更新，而是新建了一个updateStatus方法，我们在setStatus中使用它，并且可以在重试逻辑中使用。

②重新处理异步请求连接丢失时有个小问题：处理流程可能变得无序，因为ZooKeeper对请求和响应都会很好地保持顺序，但如果连接丢失，我们又再发起一个新的请求，就会导致整个时序中出现空隙。因此，我们进行一个状态更新请求前，需要先获得当前状态，否则就要放弃更新。我们通过同步方式进行检查和重试操作。

③我们执行无条件更新（第三个参数值为-1，表示禁止版本号检查），通过上下文对象参数传递状态。

①如果我们收到连接丢失的事件，我们需要用我们想要更新的状态再次调用updateStatus方法（通过setData的上下文参数传递参数），

因为在updateStatus方法中进行了竞态条件的检查，所以我们在这里就不需要再次检查。

为了更好地理解连接丢失时补发操作的问题，考虑以下场景：

- 1.从节点开始执行任务task-1，因此设置其状态为working on task-1。

- 2.客户端库尝试通过setData来实现，但此时遇到了网络问题。

- 3.客户端库确定与ZooKeeper的连接已经丢失，同时在statusUpdateCallback调用前，从节点完成了任务task-1并处于空闲状态。

- 4.从节点调用客户端库，使用setData方法置状态为Idle。

- 5.之后客户端处理连接丢失的事件，如果updateStatus方法未检查当前状态，setData调用还是会设置状态为working on task-1。

- 6.当与ZooKeeper连接重新建立时，客户端库会按顺序如实地调用这两个setData操作，这就意味着最终状态为working on task-1。

在updateStatus方法中，在补发setData之前，先检查当前状态，这样我们就可以避免以上场景。

注意：顺序和ConnectionLossException异常

ZooKeeper会严格地维护执行顺序，并提供了强有力的有序保障，然而，在多线程下还是需要小心面对顺序问题。多线程下，当回调函数中包括重试逻辑的代码时，一些常见的场景都可能导致错误发生。当遇到`ConnectionLossException`异常而补发一个请求时，新建立的请求可能排序在其他线程中的请求之后，而实际上其他线程中的请求应该在原来请求之后。

3.5 任务队列化

系统最后的组件为**Client**应用程序队列化新任务，以便从节点执行这些任务，我们会在/**tasks**节点下添加子节点来表示需要从节点需要执行的命令。我们将会使用有序节点，这样做有两个好处，第一，序列号指定了任务被队列化的顺序；第二，可以通过很少的工作为任务创建基于序列号的唯一路径。我们的**Client**代码如下：

```
import org.apache.zookeeper.ZooKeeper;
import org.apache.zookeeper.Watcher;
public class Client implements Watcher {
    ZooKeeper zk;
    String hostPort;
    Client(String hostPort) { this.hostPort = hostPort; }
    void startZK() throws Exception {
        zk = new ZooKeeper(hostPort, 15000, this);
    }
    String queueCommand(String command) throws KeeperException {
        while (true) {
            try {
                String name = zk.create("/tasks/task-", ①

                                                                                                     command.getBytes(), OPEN_ACL_UNSAFE,
                                                                                                     CreateMode.SEQUENTIAL); ②

                return name; ③

            } catch (NodeExistsException e) {
                break;
            } catch (NodeExistsException e) {
                throw new Exception(name + " already appears to be running");
            } catch (ConnectionLossException e) { ④
```

```
    }  
    }  
    public void process(WatchedEvent e) { System.out.println(e); }  
    public static void main(String args[]) throws Exception {  
        Client c = new Client(args[0]);  
        c.start();  
        String name = c.queueCommand(args[1]);  
        System.out.println("Created " + name);  
    }  
}
```

①我们在/tasks节点下创建znode节点来标识一个任务，节点名称前缀为task-。

②因为我们使用的是CreateMode.SEQUENTIAL模式的节点，task-后面会跟随一个单调递增的数字，这样就可以保证为每个任务创建的znode节点的名称是唯一的，同时ZooKeeper会确定任务的顺序。

③我们无法确定使用CreateMode.SEQUENTIAL调用create的序列号，create方法会返回新建节点的名称。

④如果我们在执行create时遇到连接丢失，我们需要重试create操作。因为多次执行创建操作，也许会为一个任务建立多个znode节点，对于大多数至少执行一次（execute-at-least-once）策略的应用程序，也没什么问题。对于某些最多执行一次（execute-at-most-once）策略的应用程序，我们就需要多一些额外工作：我们需要为每一个任务指定一个唯一的ID（如会话ID），并将其编码到znode节点名中，在遇到连

接丢失的异常时，我们只有在/**tasks**下不存在以这个会话**ID**命名的节点时才重试命令。

当我们运行**Client**应用程序并发送一个命令时，/**tasks**节点下就会创建一个新的**znode**节点，该节点并不是临时性节点，因此即使**Client**程序结束了，这个节点依然会存在。

3.6 管理客户端

最后，我们将会写一个简单的**AdminClient**，通过该程序来展示系统的运行状态。**ZooKeeper**优点之一是我们可以通过**zkCli**工具来查看系统的状态，但是通常你希望编写你自己的管理客户端，以便更快更简单地管理系统。在本例中，我们通过**getData**和**getChildren**方法来获得主从系统的运行状态。

这些方法的使用非常简单，因为这些方法不会改变系统的运行状态，我们仅需要简单地传播我们遇到的错误，而不需要进行任何清理操作。

该示例使用了同步调用的方法，这些方法还有一个**watch**参数，我们置为**false**值，因为我们不需要监视变化情况，只是想获得系统当前的运行状态。在下一章中我们将会看到如何使用这个参数来跟踪系统的变化情况。现在，让我们看一下**AdminClient**的代码：

```
import org.apache.zookeeper.ZooKeeper;
import org.apache.zookeeper.Watcher;
public class AdminClient implements Watcher {
    ZooKeeper zk;
    String hostPort;
    AdminClient(String hostPort) { this.hostPort = hostPort; }
    void start() throws Exception {
        zk = new ZooKeeper(hostPort, 15000, this);
    }
    void listState() throws KeeperException {
        try {
            Stat stat = new Stat();
            byte masterData[] = zk.getData("/master", false, stat);①
```

```

        Date startDate = new Date(stat.getCtime());②

        System.out.println("Master: " + new String(masterData) +
            " since " + startDate);
    } catch (NoNodeException e) {
        System.out.println("No Master");
    }
    System.out.println("Workers:");
    for (String w: zk.getChildren("/workers", false)) {
        byte data[] = zk.getData("/workers/" + w, false, null);③

        String state = new String(data);
        System.out.println("\t" + w + ": " + state);
    }
    System.out.println("Tasks:");
    for (String t: zk.getChildren("/assign", false)) {
        System.out.println("\t" + t);
    }
}
public void process(WatchedEvent e) { System.out.println(e); }
public static void main(String args[]) throws Exception {
    AdminClient c = new AdminClient(args[0]);
    c.start();
    c.listState();
}
}

```

①我们在/**master**节点中保存了主节点名称信息，因此我们从/**master**中获取数据，以获得当前主节点的名称。因为我们并不关心变化情况，所以我们将第二个参数置为**false**。

②我们通过**Stat**结构，可以获得当前主节点成为主节点的时间信息。ctime为该**znode**节点建立时的秒数（系统纪元以来的秒数，即自

1970年1月1日00: 00: 00UTC的描述)，详细信息请查看 `java.lang.System.currentTimeMillis`（）。

③临时节点含有两个信息：指示当前从节点正在运行；其数据表示从节点的状态。

`AdminClient`非常简单，该程序简单地通过数据结构获得在主从示例的信息。我们试着可以启停`Master`程序、`Worker`程序，多次运行`Client`来队列化一些任务，`AdminClient`会展示系统的这些变化的状态。

你也许想知道如果在`AdminClient`中使用异步API是否更好。`ZooKeeper`有一个流水线的实现，用于处理成千上万的并发请求，对于系统需要面对的各种各样的延迟问题是非常重要的，而最大的延迟在于硬盘和网络。异步和同步的组件均通过队列方式来有效利用吞吐量。`getData`方法并不会进行任何硬盘的访问，但却需要依赖网络传输，同时我们使得同步方法的请求流水线管道化。如果我们的`AdminClient`程序运行于仅有几十个从节点、几百个任务的小型系统中，也许不会是大问题，但如果系统包含的从节点、任务再上升一个数量级，延迟问题可能就变得重要了。

考虑以下场景，请求的传输往返时延时间为1毫秒，如果进程需要读取10000个`znode`节点，我们就要在网络延迟上耗费10秒的时间，而

通过异步API，我们可以缩短整个时间到接近1秒的时间。

以上面的Master、Worker、Client这些的基本实现带领我们进入了主从系统的开端，但到目前为止还没有实际调度起来。当一个任务加入队列，主节点需要唤醒并分配任务给一个从节点，从节点需要找出分配给自己的任务，任务完成时，客户端需要及时知道，如果主节点故障，另一个等待中的主节点需要接管主节点工作。如果从节点故障，分配给这个从节点的任务需要分配给其他从节点，在下一章中，我们将会讨论这些必要功能的实现。

3.7 小结

我们在zkCli工具中使用的命令与我们通过ZooKeeper编程所使用的API非常接近，因此，zkCli工具在初期调研时非常有用，我们可以通过该工具尝试不同的应用数据的组织方式。API与zkCli的命令非常接近，通过zkCli工具调研后，我们就可以快速写出与zkCli命令对应的应用程序。然而还有一些注意事项。首先，我们常常在一个稳定环境中使用zkCli工具，而不会发生某些未知故障，而对于需要部署的代码，我们需要处理异常情况使得我们的代码非常复杂，尤其是ConnectionLossException异常，开发者需要检查系统状态并合理恢复（ZooKeeper用于协助管理分布式状态，提供了故障处理的框架，但遗憾的是，它并不能让故障消失）。其次，适应异步API的开发非常有用，异步API提供了巨大的性能优势，简化了错误恢复工作。

第4章 处理状态变化

在应用程序中，需要知道ZooKeeper集合的状态，这种情况并不少见。例如，在第1章的例子中，备份主节点需要知道主要主节点已经崩溃，从节点需要知道任务分配给了自己，甚至ZooKeeper的客户端会定时轮询ZooKeeper集合，检查系统状态是否发生了变化。然而轮询方式并非高效的方式，尤其是在期望的变化发生频率很低时。

举例说明，在主要主节点崩溃时，备份主节点需要知道这一情况，以便它们可以进行故障处理。为了减少主节点崩溃后的恢复时间，我们需要频繁轮询，如每50毫秒，只是为了用示例说明积极轮询的情况。在这种情况下，每个备份主节点每秒会产生20个请求，如果有多个备份主节点，备份主节点的数量乘上这个频率，就得到了为得到主要主节点状态而轮询ZooKeeper所消耗的全部请求量，即使像ZooKeeper这样的系统处理这个数量请求也非常简单，但主要主节点崩溃的情况很少发生，因此这些请求其实是多余的。假设我们通过增加主节点状态查询的轮询周期来减少对ZooKeeper的查询数量，如1秒，周期时间的增加则会导致主节点崩溃时恢复时间的增加。

我们完全可以通过ZooKeeper通知客户端感兴趣的具体事件来避免轮询的调优和轮询流量。ZooKeeper提供了处理变化的重要机制——监视点（watch）。通过监视点，客户端可以对指定的znode节点注册一

个通知请求，在发生变化时就会收到一个单次的通知。例如，我们的主要主节点创建了一个临时性的**znode**节点来标识主节点锁，而备份主节点注册一个监视点来监视这个主节点锁是否存在，如果主节点崩溃，主节点锁自动被删除，并通知所有备份主节点。一旦备份主节点收到通知，它们就可以开始进行主节点选举，如3.3节中所述，通过尝试创建一个临时的**znode**节点来标识主节点锁。

监视点和通知形成了一个通用机制，使客户端可以观察变化情况，而不用不断地轮询**ZooKeeper**。我们已经通过主节点的例子进行了说明，但该通用机制还适用于很多情况。

4.1 单次触发器

在深入讨论监视点之前，我们先了解一些术语。我们所说的事件（**event**）表示一个**znode**节点执行了更新操作。而一个监视点（**watch**）表示一个与之关联的**znode**节点和事件类型组成的单次触发器（例如，**znode**节点的数据被赋值，或**znode**节点被删除）。当一个监视点被一个事件触发时，就会产生一个通知（**notification**）。通知是注册了监视点的应用客户端收到的事件报告的消息。

当应用程序注册了一个监视点来接收通知，匹配该监视点条件的第一个事件会触发监视点的通知，并且最多只触发一次。例如，当**znode**节点/**z**被删除，客户端需要知道该变化（例如，表示备份主节点），客户端在/**z**节点执行**exists**操作并设置监视点标志位，等待通知，客户端会以回调函数的形式收到通知。

客户端设置的每个监视点与会话关联，如果会话过期，等待中的监视点将会被删除。不过监视点可以跨越不同服务端的连接而保持，例如，当一个**ZooKeeper**客户端与一个**ZooKeeper**服务端的连接断开后连接到集合中的另一个服务端，客户端会发送未触发的监视点列表，在注册监视点时，服务端将要检查已监视的**znode**节点在之前注册监视

点之后是否已经变化，如果znode节点已经发生变化，一个监视点的事件就会被发送给客户端，否则在新的服务端上注册监视点。

单次触发是否会丢失事件

答案是肯定的。一个应用在接收到通知后，注册另一个监视点时，可能会丢失事件，不过，这个问题需要再深入讨论。丢失事件通常并不是问题，因为任何在接收通知与注册新监视点之间的变化情况，均可以通过读取ZooKeeper的状态信息来获得。

假设一个从节点接收到一个新任务分配给它的通知。为了接收新任务，从节点读取任务列表，如果在通知接收后，又给这个从节点分配了更多的任务，在通过getChildren调用获取任务列表时会返回所有的任务。同时调用getChildren时也可以设置新的监视点，从而保证从节点不会丢失任务。

实际上，将多个事件分摊到一个通知上具有积极的作用，比如，应用进行高频率的更新操作时，这种通知机制比每个事件都发送通知更加轻量化。举个例子，如果每个通知平均捕获两个事件，我们为每个事件只产生了0.5个通知，而不是每个事件1个通知。

4.2 如何设置监视点

ZooKeeper的API中的所有读操作：`getData`、`getChildren`和`exists`，均可以选择在读取的`znode`节点上设置监视点。使用监视点机制，我们需要实现`Watcher`接口类，实现其中的`process`方法：

```
public void process(WatchedEvent event);
```

`WatchedEvent`数据结构包括以下信息：

- ZooKeeper会话状态（`KeeperState`）：`Disconnected`、`SyncConnected`、`AuthFailed`、`ConnectedReadOnly`、`SaslAuthenticated`和`Expired`。

- 事件类型（`EventType`）：`NodeCreated`、`NodeDeleted`、`NodeDataChanged`、`NodeChildrenChanged`和`None`。

- 如果事件类型不是`None`时，返回一个`znode`路径。

其中前三个事件类型只涉及单个`znode`节点，第四个事件类型涉及监视的`znode`节点的子节点。我们使用`None`表示无事件发生，而是ZooKeeper的会话状态发生了变化。

监视点有两种类型：数据监视点和子节点监视点。创建、删除或设置一个znode节点的数据都会触发数据监视点，exists和getData这两个操作可以设置数据监视点。只有getChildren操作可以设置子节点监视点，这种监视点只有在znode子节点创建或删除时才被触发。对于每种事件类型，我们通过以下调用设置监视点：

NodeCreated

通过exists调用设置一个监视点。

NodeDeleted

通过exists或getData调用设置监视点。

NodeDataChanged

通过exists或getData调用设置监视点。

NodeChildrenChanged

通过getChildren调用设置监视点。

当创建一个ZooKeeper对象（见第3章），我们需要传递一个默认的Watcher对象，ZooKeeper客户端使用这个监视点来通知应用ZooKeeper状态的变化情况，如会话状态的变化。对于ZooKeeper节点

的事件的通知，你可以使用默认的监视点，也可以单独实现一个。例如，`getData`调用有两种方式设置监视点：

```
public byte[] getData(final String path, Watcher watcher, Stat stat);  
public byte[] getData(String path, boolean watch, Stat stat);
```

两个方法第一个参数均为`znode`节点，第一个方法传递一个新的`Watcher`对象（我们已经创建完毕），第二个方法则告诉客户端使用默认的监视点，我们只需要在调用时将第二个参数传递`true`。

`stat`入参为`Stat`类型的实例化对象，`ZooKeeper`使用该对象返回指定的`path`参数的`znode`节点信息。`Stat`结构包括`znode`节点的属性信息，如该`znode`节点的上次更新（`zxid`）的时间戳，以及该`znode`节点的子节点数。

对于监视点的一个重要问题是，一旦设置监视点就无法移除。要想移除一个监视点，只有两个方法，一是触发这个监视点，二是使其会话被关闭或过期。在未来版本中可能会改变这个特性，这是因为开发社区致力于在版本3.5.0中提供该功能。

[注意：关于一些重载](#)

在`ZooKeeper`的会话状态和`znode`节点的变化事件中，我们使用了相同的监视机制来处理应用程序的相关事件的通知。虽然会话状态的

变化和**znode**状态的变化组成了两个独立的事件集合，为简单起见，我们使用了相同的机制传送这些事件。

4.3 普遍模型

我们进入主-从模式例子的章节之前，先看一些ZooKeeper的应用中使用的通用代码的模型：

- 1.进行调用异步。
- 2.实现回调对象，并传入异步调用函数中。
- 3.如果操作需要设置监视点，实现一个Watcher对象，并传入异步调用函数中。

以下为exists的异步调用的示例代码：

```
zk.exists("/myZnode", ①

    myWatcher,
    existsCallback,
    null);
Watcher myWatcher = new Watcher() {②

    public void process(WatchedEvent e) {
        // Process the watch event
    }
}
StatCallback existsCallback = new StatCallback() {③
```

```
    public void processResult(int rc, String path, Object ctx, Stat stat) {  
        // Process the result of the exists call  
    }  
};
```

①ZooKeeper的exists调用，注意该调用为异步方式。

②Watcher的实现。

③exists的回调对象。

之后我们会看到，该框架使用非常广泛。

4.4 主-从模式的例子

现在，我们通过主-从模式的例子来看看如何处理状态的变化。以下为一个任务列表，一个组件需要等待处理的变化情况：

- 管理权变化。
- 主节点等待从节点列表的变化。
- 主节点等待新任务进行分配。
- 从节点等待分配新任务。
- 客户端等待任务的执行结果。

我们之后会通过一些代码片段来说明如何在ZooKeeper环境中处理这些任务。

本书附件部分提供了完整的实例代码。

4.4.1 管理权变化

回忆在3.3节讨论的内容，应用客户端通过创建/master节点来推选自己为主节点（我们称为“主节点竞选”），如果znode节点已经存在，

应用客户端确认自己不是主要主节点并返回，然而，这种实现方式无法容忍主要主节点的崩溃。如果主要主节点崩溃，备份主节点并不知道，因此我们需要在/master上设置监视点，在节点删除时（无论是显式关闭还是因为主要主节点的会话过期），ZooKeeper会通知客户端。

为了设置监视点，我们新建一个新的监视点对象，命名为masterExistsWatcher，并传入exists方法中。一旦/master删除就会发出通知，就会调用在masterExistsWatcher定义的process函数，并调用runForMaster方法。

```
StringCallback masterCreateCallback = new StringCallback() {
    public void processResult(int rc, String path, Object ctx, String name) {
        switch (Code.get(rc)) {
            case CONNECTIONLOSS:
                checkMaster();①

                break;
            case OK:
                state = MasterStates.ELECTED;
                takeLeadership();②

                break;
            case NODEEXISTS:
                state = MasterStates.NOTELECTED;
                masterExists();③

                break;
            default:
                state = MasterStates.NOTELECTED;
                LOG.error("Something went wrong when running for master.",④
```

```

        KeeperException.create(Code.get(rc), path));
    }
};
void masterExists() {
    zk.exists("/master", ⑤

        masterExistsWatcher,
        masterExistsCallback,
        null);
}
Watcher masterExistsWatcher = new Watcher() {
    public void process(WatchedEvent e) {
        if(e.getType() == EventType.NodeDeleted) {
            assert "/master".equals( e.getPath() );
            runForMaster();⑥
        }
    }
};

```

①在连接丢失事件发生的情况下，客户端检查/**master**节点是否存在，因为客户端并不知道是否能够创建这个节点。

②如果返回**OK**，那么开始行使领导权。

③如果其他进程已经创建了这个**znode**节点，客户端需要监视该节点。

④如果发生了某些意外情况，就会记录错误日志，而不再做其他事情。

⑤通过**exists**调用在/**master**节点上设置了监视点。

⑥如果/master节点删除了，那么再次竞选主节点。

下面继续采用我们在3.3.1节中所讨论的异步方式，我们同样需要为exists调用创建一个回调函数，以便在回调函数中关注某些情况。首先，在发生连接丢失的事件时，因为需要在/master节点上设置监视点，所以需要再次调用exists操作；其次，在create的回调方法执行和exists操作执行之间发生了/master节点被删除的情况，因此在exists返回操作成功后（返回OK），我们需要检查返回的stat对象是否为空，因为当节点不存在时，stat为null；最后，如果返回的结果不是OK或CONNECTIONLOSS，我们通过获取节点数据来检查/master节点。加入客户端的会话过期，在这种情况下，获得/master数据的回调方法会记录一个错误信息并退出。以下为我们的exists回调方法的代码：

```
StatCallback masterExistsCallback = new StatCallback() {
    public void processResult(int rc, String path, Object ctx, Stat stat) {
        switch (Code.get(rc)) {
            case CONNECTIONLOSS:
                masterExists();①

                break;
            case OK:
                if(stat == null) {
                    state = MasterStates.RUNNING;
                    runForMaster();②
                }
                break;
            default:
                checkMaster();③
        }
    }
}
```

```
        break;
    }
};
```

①连接丢失的情况下重试。

②如果返回OK，判断znode节点是否存在，不存在就竞选主节点。

③如果发生意外情况，通过获取节点数据来检查/master是否存在。

对/master节点执行exists操作，返回结果也许是该znode节点已经被删除，这时因为无法保证监视点的设置是在znode节点删除前，所以客户端需要再次竞选/master节点。如果再次尝试成为主节点失败，那么客户端就知道有其他客户端成功了，之后该客户端就需要再次为/master节点添加监视点。如果收到的是/master节点创建的通知，而不是删除通知，客户端就不再竞选/master节点，同时，对应的exists操作（设置监视点的操作）会返回/master节点不存在，然后触发exists回调方法，进行/master节点竞选操作。

我们注意到，只要客户端程序运行中，且没有成为主要主节点，客户端竞选主节点并执行exists来设置监视点，这一模式就会一直运行下去。如果客户端成为主要主节点，但却崩溃了，客户端重启还继续重新执行这些代码。

图4-1直观地展示了这些交错的操作。如果竞选主节点成功（图中a），`create`操作执行完成，应用客户端不需要做其他事情。如果`create`操作失败，则意味着该节点已经存在，客户端就会执行`exists`操作来设置`/master`节点的监视点（图中b）。在竞选主节点和执行`exists`操作之间，也许`/master`节点已经删除了，这时，如果`exists`调用返回该节点依然存在，客户端只需要等待通知的到来，否则就需要再次尝试创建`/master`进行竞选主节点操作。如果创建`/master`节点成功，监视点就会被触发，表示`znode`节点发生了变化（图中c），不过，这个通知没有什么意义，因为这是客户端自己引起的变化。如果再次执行`create`操作失败，我们就会通过执行`exists`设置监视点来重新执行这一流程（图中d）。

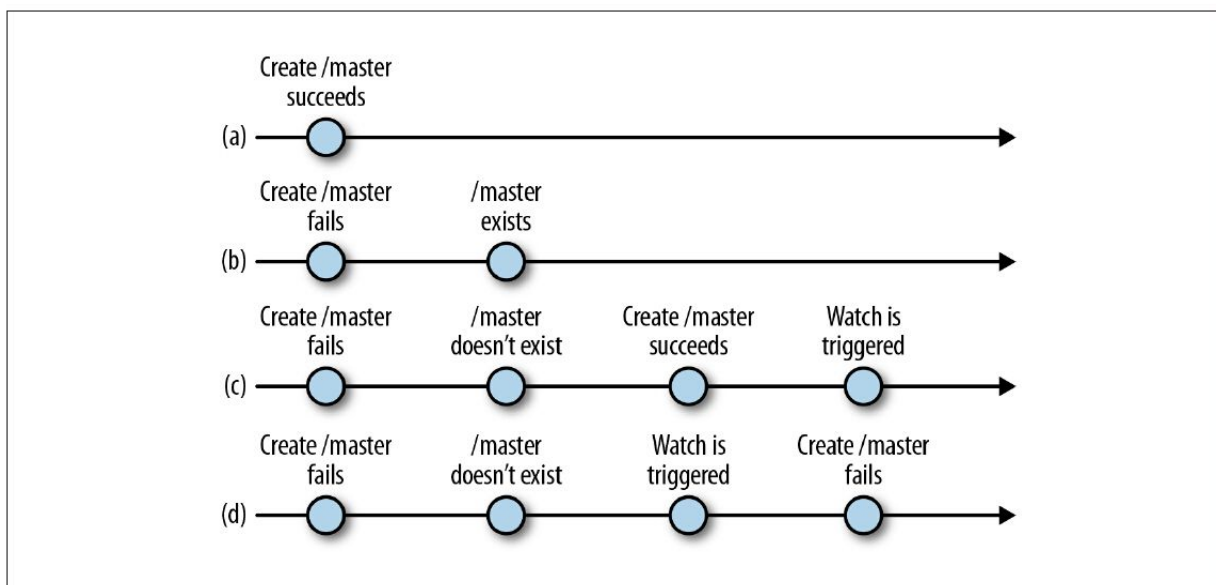


图4-1：主节点竞选中可能的交错操作

4.4.2 主节点等待从节点列表的变化

系统中任何时候都可能发生新的从节点加入进来，或旧的从节点退役的情况，从节点执行分配给它的任务前也许会崩溃。为了确认某个时间点可用的从节点信息，我们通过在ZooKeeper中的/workers下添加子节点来注册新的从节点。当一个从节点崩溃或从系统中被移除，如会话过期等情况，需要自动将对应的znode节点删除。优雅实现的从节点会显式地关闭其会话，而不需要ZooKeeper等待会话过期。

主要主节点使用getChildren来获取有效的从节点列表，同时还会监控这个列表的变化。以下为获取列表并监视变化的示例代码：

```
Watcher workersChangeWatcher = new Watcher() {①

    public void process(WatchedEvent e) {
        if(e.getType() == EventType.NodeChildrenChanged) {
            assert "/workers".equals( e.getPath() );
            getWorkers();
        }
    }
};

void getWorkers() {
    zk.getChildren("/workers",
                   workersChangeWatcher,
                   workersGetChildrenCallback,
                   null);
}

ChildrenCallback workersGetChildrenCallback = new ChildrenCallback() {
    public void processResult(int rc, String path, Object ctx,
                             List<String> children) {
        switch (Code.get(rc)) {
            case CONNECTIONLOSS:
                getWorkerList();②
        }
    }
};
```

```

        break;
    case OK:
        LOG.info("Succesfully got a list of workers: "
            + children.size()
            + " workers");
        reassignAndSet(children);③

        break;
    default:
        LOG.error("getChildren failed",
            KeeperException.create(Code.get(rc), path));
    }
}
};

```

①workersChangeWatcher为从节点列表的监视点对象。

②当CONNECTIONLOSS事件发生时，我们需要重新获取子节点并设置监视点的操作。

③重新分配崩溃从节点的任务，并重新设置新的从节点列表。

我们从getWorkerList方法开始执行，通过异步方式执行getChildren方法，传入workersGetChildrenCallback参数用于处理操作结果。如果客户端失去与服务端的连接（CONNECTIONLOSS事件），监视点不会被添加，我们也不会得到从节点的列表，我们再次执行getWorkerList来设置监视点并获取从节点列表，如果执行getChildren成功，我们会调用reassignAndSet方法，该方法的代码如下：

```

ChildrenCache workersCache;①

```

```

void reassignAndSet(List<String> children) {
    List<String> toProcess;
    if(workersCache == null) {
        workersCache = new ChildrenCache(children);②

        toProcess = null;③

    } else {
        LOG.info( "Removing and setting" );
        toProcess = workersCache.removedAndSet( children );④
    }
    if(toProcess != null) {
        for(String worker : toProcess) {
            getAbsentWorkerTasks(worker);⑤
        }
    }
}

```

①该变量用于保存上次获得的从节点列表的本地缓存。

②如果第一次使用本地缓存这个变量，那么初始化该变量。

③第一次获得所有从节点时，不需要做什么其他事。

④如果不是第一次，那么需要检查是否有从节点已经被移除了。

⑤如果有从节点被移除了，我们就需要重新分配任务。

我们需要保存之前获得的信息，因此使用本地缓存。假设在我们第一次获得从节点列表后，当收到从节点列表更新的通知时，如果我们没有保存旧的信息，即使我们再次读取信息也不知道具体变化的信息是什么。本例中的缓存类简单地保存主节点上次读取的列表信息，并实现检查变化信息的一些方法。

注意：基于CONNECTIONLOSS事件的监视

监视点的操作执行成功后就会为一个znode节点设置一个监视点，如果ZooKeeper的操作因为客户端连接断开而失败，应用需要再次执行这些调用。

4.4.3 主节点等待新任务进行分配

与等待从节点列表变化类似，主要主节点等待添加到/tasks节点中的新任务。主节点首先获得当前的任务集，并设置变化情况的监视点。在ZooKeeper中，/tasks的子节点表示任务集，每个子节点对应一个任务，一旦主节点获得还未分配的任务信息，主节点会随机选择一个从节点，将这个任务分配给从节点。以下assignTasks方法为任务分配的实现：

```
Watcher tasksChangeWatcher = new Watcher() {①
```

```

        public void process(WatchedEvent e) {
            if(e.getType() == EventType.NodeChildrenChanged) {
                assert "/tasks".equals( e.getPath() );
                getTasks();
            }
        }
    };
    void getTasks() {
        zk.getChildren("/tasks",
            tasksChangeWatcher,
            tasksGetChildrenCallback,
            null);②
    }

    ChildrenCallback tasksGetChildrenCallback = new ChildrenCallback() {
        public void processResult(int rc,
            String path,
            Object ctx,
            List<String> children) {
            switch(Code.get(rc)) {
                case CONNECTIONLOSS:
                    getTasks();
                    break;
                case OK:
                    if(children != null) {
                        assignTasks(children);③
                    }
                    break;
                default:
                    LOG.error("getChildren failed.",
                        KeeperException.create(Code.get(rc), path));
            }
        }
    };
};

```

①在任务列表变化时，处理通知的监视点实现。

②获得任务列表。

③分配列表中的任务。

现在我们实现`assignTasks`方法，该方法简单地分配在`/tasks`子节点表示的列表中的每个任务。在建立任务分配的`znode`节点前，我们首先通过`getData`方法获得任务信息：

```
void assignTasks(List<String> tasks) {
    for(String task : tasks) {
        getTaskData(task);
    }
}

void getTaskData(String task) {
    zk.getData("/tasks/" + task,
        false,
        taskDataCallback,
        task);①

}

DataCallback taskDataCallback = new DataCallback() {
    public void processResult(int rc,
                             String path,
                             Object ctx,
                             byte[] data,
                             Stat stat) {
        switch(Code.get(rc)) {
            case CONNECTIONLOSS:
                getTaskData((String) ctx);
                break;
            case OK:
                /*
                 * Choose worker at random.
                 */
                int worker = rand.nextInt(workerList.size());
                String designatedWorker = workerList.get(worker);
                /*
                 * Assign task to randomly chosen worker.
                 */
                String assignmentPath = "/assign/" + designatedWorker + "/" +
                    (String) ctx;
                createAssignment(assignmentPath, data);②

                break;
            default:
                LOG.error("Error when trying to get task data.",
                    KeeperException.create(Code.get(rc), path));
        }
    }
};
```

①获得任务信息。

②随机选择一个从节点，分配任务给这个从节点。

首先我们需要获得任务的信息，因为分配任务后，我们会删除/tasks下的这个子节点，这样，主节点就不需要记住分配了哪些任务。让我们看一下分配一个任务的代码：

```
void createAssignment(String path, byte[] data) {
    zk.create(path,
        data, Ids.OPEN_ACL_UNSAFE,
        CreateMode.PERSISTENT,
        assignTaskCallback,
        data);①

}
StringCallback assignTaskCallback = new StringCallback() {
    public void processResult(int rc, String path, Object ctx, String name) {
        switch(Code.get(rc)) {
            case CONNECTIONLOSS:
                createAssignment(path, (byte[]) ctx);
                break;
            case OK:
                LOG.info("Task assigned correctly: " + name);
                deleteTask(name.substring( name.lastIndexOf("/") + 1 ));②

                break;
            case NODEEXISTS:
                LOG.warn("Task already assigned");
                break;
            default:
                LOG.error("Error when trying to assign task.",
                    KeeperException.create(Code.get(rc), path));
        }
    }
};
```

①创建分配节点，路径形式为/assign/worker-id/task-num。

②删除/tasks下对应的任务节点。

对于新任务，主节点选择一个从节点分配任务之后，主节点就会在/assign/work-id节点下创建一个新的znode节点，其中id为从节点标识符，之后主节点从任务列表中删除该任务节点。上面的例子中，其中删除znode节点的代码采用了我们之前讲到的模式的代码。

当主节点为某个标识符为id的从节点创建任务分配节点时，假设从节点在任务分配节点（/assign/work-id）上注册了监视点，ZooKeeper会向从节点发送一个通知。

注意，主节点在成功分配任务后，会删除/tasks节点下对应的任务。这种方式简化了主节点角色接收新任务并分配的设计，如果任务列表中混合的已分配和未分配的任务，主节点还需要区分这些任务。

4.4.4 从节点等待分配新任务

从节点第一步需要先向ZooKeeper注册自己，如我们之前已经讨论过的，在/workers节点下创建一个子节点：

```
void register() {
    zk.create("/workers/worker-" + serverId,
              new byte[0],
              Ids.OPEN_ACL_UNSAFE,
              CreateMode.EPHEMERAL,
              createWorkerCallback, null);①
```

```

}
StringCallback createWorkerCallback = new StringCallback() {
    public void processResult(int rc, String path, Object ctx, String name) {
        switch (Code.get(rc)) {
            case CONNECTIONLOSS:
                register(); ②
                break;
            case OK:
                LOG.info("Registered successfully: " + serverId);
                break;
            case NODEEXISTS:
                LOG.warn("Already registered: " + serverId);
                break;
            default:
                LOG.error("Something went wrong: " +
                    KeeperException.create(Code.get(rc), path));
        }
    }
};

```

①通过创建一个**znode**节点来注册从节点。

②重试，注意再次注册不会有问题，因为如果**znode**节点已经存在，我们会收到**NODEEXISTS**事件。

添加该**znode**节点会通知主节点，这个从节点的状态是活跃的，且已准备好处理任务。注意我们并未使用第3章中介绍的空闲/忙碌（**idle/busy**）状态，只是为了该示例的简化。

同样，我们还创建了**/assign/work-id**节点，这样，主节点可以为这个从节点分配任务。如果我们在创建**/assign/work-id**节点之前创建了**/workers/worker-id**节点，我们可能会陷入以下情况，主节点尝试分配任务，因分配节点的父节点还没有创建，导致主节点分配失败。为了

避免这种情况，我们需要先创建/assign/worker-id节点，而且从节点需要在/assign/worker-id节点上设置监视点来接收新任务分配的通知。

一旦有任务列表分配给从节点，从节点就会从/assign/worker-id获取任务信息并执行任务。从节点从本地列表中获取每个任务的信息并验证任务是否还在待执行的队列中，从节点保存一个本地待执行任务的列表就是为了这个目的。注意，为了释放回调方法的线程，我们在单独的线程对从节点的已分配任务进行循环，否则，我们会阻塞其他的回调方法的执行。示例中，我们使用了Java的ThreadPoolExecutor类分配一个线程，该线程进行任务的循环操作：

```
Watcher newTaskWatcher = new Watcher() {
    public void process(WatchedEvent e) {
        if(e.getType() == EventType.NodeChildrenChanged) {
            assert new String("/assign/worker-"+ serverId).equals( e.getPath() );
            getTasks();①
        }
    }
};
void getTasks() {
    zk.getChildren("/assign/worker-" + serverId,
        newTaskWatcher,
        tasksGetChildrenCallback,
        null);
}
ChildrenCallback tasksGetChildrenCallback = new ChildrenCallback() {
    public void processResult(int rc,
        String path,
        Object ctx,
        List<String> children) {
        switch(Code.get(rc)) {
            case CONNECTIONLOSS:
                getTasks();
                break;
            case OK:
                if(children != null) {
                    executor.execute(new Runnable() {②
```

```

        List<String> children;
        DataCallback cb;
        public Runnable init (List<String> children,
                               DataCallback cb) {
            this.children = children;
            this.cb = cb;
            return this;
        }
        public void run() {
            LOG.info("Looping into tasks");
            synchronized(onGoingTasks) {
                for(String task : children) {③

                    if(!onGoingTasks.contains( task )) {
                        LOG.trace("New task: {}", task);
                        zk.getData("/assign/worker-" +
                                   serverId + "/" + task,
                                   false,
                                   cb,
                                   task);④

                        onGoingTasks.add( task );⑤
                    }
                }
            }
        }
        .init(children, taskDataCallback));
    }
    break;
default:
    System.out.println("getChildren failed: " +
                        KeeperException.create(Code.get(rc), path));
}
}
};

```

①当收到子节点变化的通知后，获得子节点的列表。

- ②单独线程中执行。
- ③循环子节点列表。
- ④获得任务信息并执行任务。
- ⑤将正在执行的任务添加到执行中列表，防止多次执行。

注意：会话事件和监视点

当我们与服务端的连接断开时（例如，服务端崩溃时），直到连接重新建立前，不会传送任何监视点。因此，会话事件 **CONNECTIONLOSS** 会发送给所有已知的监视点进行处理。一般来说，应用使用会话事件进入安全模式：**ZooKeeper**客户端在失去连接后不会接收任何事件，因此客户端需要继续保持这种状态。在我们的主从应用的例子中，处理提交任务外，其他所有动作都是被动的，所以如果主节点或从节点发生连接断开时，不会触发任何动作。而且在连接断开时，主从应用中的客户端也无法提交新任务以及接收任务状态的通知。

4.4.5 客户端等待任务的执行结果

假设应用客户端已经提交了一个任务，现在客户端需要知道该任务何时被执行，以及任务状态。回忆之前所讨论的，从节点执行执行

一个任务时，会在/status下创建一个znode节点。让我们先看下提交任务执行的代码：

```
void submitTask(String task, TaskObject taskCtx) {
    taskCtx.setTask(task);
    zk.create("/tasks/task-",
        task.getBytes(),
        Ids.OPEN_ACL_UNSAFE,
        CreateMode.PERSISTENT_SEQUENTIAL,
        createTaskCallback,
        taskCtx);①
}

StringCallback createTaskCallback = new StringCallback() {
    public void processResult(int rc, String path, Object ctx, String name) {
        switch (Code.get(rc)) {
            case CONNECTIONLOSS:
                submitTask(((TaskObject) ctx).getTask(),
                    (TaskObject) ctx);②

                break;
            case OK:
                LOG.info("My created task name: " + name);
                ((TaskObject) ctx).setTaskName(name);
                watchStatus("/status/" + name.replace("/tasks/", ""),
                    ctx);③

                break;
            default:
                LOG.error("Something went wrong" +
                    KeeperException.create(Code.get(rc), path));
        }
    }
};
```

①与之前的ZooKeeper调用不同，我们传递了一个上下文对象，该对象为我们实现的Task类的实例。

②连接丢失时，再次提交任务，注意重新提交任务可能会导致任务重复。

③为这个任务的znode节点设置一个监视点。

注意：有序节点是否创建成功？

在创建有序节点时发生CONNECTIONLOSS事件，处理这种情况比较棘手，因为ZooKeeper每次分配一个序列号，对于连接断开的客户端，无法确认这个节点是否创建成功，尤其是在其他客户端一同并发请求时（我们提到的并发请求是指多个客户端进行相同的请求）。为了解决这个问题，我们需要添加一些提示信息来标记这个znode节点的创建者，比如，在任务名称中加入服务器ID等信息，通过这个方法，我们就可以通过获取所有任务列表来确认任务是否添加成功。

我们检查状态节点是否已经存在（也许任务很快处理完成），并设置监视点。我们提供了一个收到znode节点创建的通知时进行处理的监视点的实现和一个exists方法的回调实现：

```
ConcurrentHashMap<String, Object> ctxMap =
    new ConcurrentHashMap<String, Object>();
void watchStatus(String path, Object ctx) {
    ctxMap.put(path, ctx);
    zk.exists(path,
        statusWatcher,
        existsCallback,
        ctx);①
}
```

```

Watcher statusWatcher = new Watcher() {
    public void process(WatchedEvent e) {
        if(e.getType() == EventType.NodeCreated) {
            assert e.getPath().contains("/status/task-");
            zk.getData(e.getPath(),
                false,
                getDataCallback,
                ctxMap.get(e.getPath()));
        }
    }
};
StatCallback existsCallback = new StatCallback() {
    public void processResult(int rc, String path, Object ctx, Stat stat) {
        switch (Code.get(rc)) {
            case CONNECTIONLOSS:
                watchStatus(path, ctx);
                break;
            case OK:
                if(stat != null) {
                    zk.getData(path, false, getDataCallback, null);②
                }
                break;
            case NONODE:
                break;③

            default:
                LOG.error("Something went wrong when " +
                    "checking if the status node exists: " +
                    KeeperException.create(Code.get(rc), path));
                break;
        }
    }
};

```

①客户端通过该方法传递上下对象，当收到状态节点的通知时，就可以修改这个表示任务的对象（TaskObject）。

②状态节点已经存在，因此客户端获取这个节点信息。

③如果状态节点不存在，这是常见情况，客户端不进行任何操作。

4.5 另一种调用方式: Multiop

Multiop并非ZooKeeper的原始设计，该特性在3.4.0版本中被添加进来。**Multiop**可以原子性地执行多个ZooKeeper的操作，执行过程为原子性，即在**multiop**代码块中的所有操作要不全部成功，要不全部失败。例如，我们在一个**multiop**块中删除一个父节点以及其子节点，执行结果只可能是这两个操作都成功或都失败，而不可能是父节点被删除而子节点还存在，或子节点被删除而父节点还存在。

使用**multiop**特性:

- 1.创建一个Op对象，该对象表示你想通过**multiop**方法执行的每个ZooKeeper操作，ZooKeeper提供了每个改变状态操作的Op对象的实现: `create`、`delete`和`setData`。

- 2.通过Op对象中提供的一个静态方法调用进行操作。

- 3.将Op对象添加到Java的Iterable类型对象中，如列表（List）。

- 4.使用列表对象调用**multi**方法。

以下示例说明这个过程:

```
Op deleteZnode(String z) {①
```

```
        return Op.delete(z, -1);②

    }
    ...
    List<OpResult> results = zk.multi(Arrays.asList(deleteZnode("/a/b"),
                                                    deleteZnode("/a"));③
```

①为delete方法创建Op对象。

②通过对应的Op方法返回对象。

③以列表方式传入每个delete操作的元素执行multi方法。

调用multi方法返回一个OpResult对象的列表，每个对象对应每个操作。例如，对于delete操作，我们使用DeleteResult类，该类继承自OpResult，通过每种操作类型对应的结果对象暴露方法和数据。

DeleteResult对象仅提供了equals和hashCode方法，而CreateResult对象暴露出操作的路径（path）和Stat对象。对于错误处理，ZooKeeper返回一个包含错误码的ErrorResult类的实例。

multi方法同样也有异步版本，以下为同步方法和异步方法的定义：

```
public List<OpResult> multi(Iterable<Op> ops) throws InterruptedException,  
                                                                    KeeperException;  
public void multi(Iterable<Op> ops, MultiCallback cb, Object ctx);
```

Transaction封装了**multi**方法，提供了简单的接口。我们可以创建**Transaction**对象的实例，添加操作，提交事务。使用**Transaction**重写上例的代码如下：

```
Transaction t = new Transaction();  
t.delete("/a/b", -1);  
t.delete("/a", -1);  
List<OpResult> results = t.commit();
```

commit方法同样也有一个异步版本的方法，该方法以**MultiCallback**对象和上下文对象为输入：

```
public void commit(MultiCallback cb, Object ctx);
```

multiop可以简化不止一处的主从模式的实现，当分配一个任务，在之前的例子中，主节点会创建任务分配节点，然后删除/**tasks**下对应的任务节点。如果在删除/**tasks**下的节点时，主节点崩溃，就会导致一个已分配的任务还在/**tasks**下。使用**multiop**，我们可以原子化创建任务分配节点和删除/**tasks**下对应的任务节点这两个操作。使用这个方式，我们可以保证没有已分配的任务还在/**tasks**节点下，如果备份节点接管了主节点角色，就不用再区分/**tasks**下的任务是不是没有分配的。

`multiop`提供的另一个功能是检查一个`znode`节点版本，通过`multiop`可以同时读取的多个节点的`ZooKeeper`状态并回写数据——如回写某些读取到的数据信息。当被检查的`znode`版本号没有变化时，就可以通过`multiop`调用来检查没有被修改的`znode`节点版本号，这个功能非常有用，如在检查一个或多个`znode`节点版本号取决于另外一个`znode`节点版本号时。在我们的主从模式的示例中，主节点需要让客户端在主节点指定的路径下添加新任务，例如，主节点要求客户端在`/task-mid`的子节点中添加新任务节点，其中`mid`为主节点的标识符，主节点在`/master-path`节点中保存这个路径的数据，客户端在添加新任务前，需要先读取`/master-path`的数据，并通过`Stat`获取这个节点版本号信息，然后，客户端通过`multiop`的部分调用方式在`/task-mid`节点下添加新任务节点，同时会检查`/master-path`版本号是否与之前读取的相匹配。

`check`方法的定义与`setData`方法相似，只是没有`data`参数：

```
public static Op check(String path, int version);
```

如果输入的`path`的`znode`节点版本号不匹配，`multi`调用会失败。通过以下简单的示例代码，我们来说明如何实现上面所讨论的场景：

```
byte[] masterData = zk.getData("/master-path", false, stat);①
```

```
String parent = new String(masterData);②
```

```
...  
zk.multi(Arrays.asList(Op.check("/master-path", stat.getVersion()),  
                        Op.create(, modify(z1Data), -1), ③
```

①获取/**master**节点的数据。

②从/**master**节点获得路径信息。

③两个操作的**multi**调用。

注意，如果我们在/**master**节点中以主节点ID来保存路径信息，以上方式就无法正常运行，因为新主节点每次都会创建/**master**，从而导致/**master**的版本号始终为1。

4.6 通过监视点代替显式缓存管理

从应用的角度来看，客户端每次都是通过访问ZooKeeper来获取给定znode节点的数据、一个znode节点的子节点列表或其他相关的ZooKeeper状态，这种方式并不可取。反而更高效的方式为客户端本地缓存数据，并在需要时使用这些数据，一旦这些数据发生变化，你让ZooKeeper通知客户端，客户端就可以更新缓存的数据。这些通知与我们之前所讨论的一样，应用的客户端通过注册监视点来接收这些通知消息。总之，监视点可以让客户端在本地缓存一个版本的数据（比如，一个znode节点数据或节点的子节点列表信息），并在数据发生变化时接收到通知来进行更新。

ZooKeeper的设计者还可以采用另一种方式，客户端透明地缓存客户端访问的所有ZooKeeper状态，并在更新缓存数据时将这些数据置为无效。实现这种缓存一致性的方案代价非常大，因为客户端也许并不需要缓存所有它们所访问的ZooKeeper状态，而且服务端需要将缓存状态置为无效，为了实现失效机制，服务端不得不关注每个客户端中缓存的信息，并广播失效请求。客户端数量很大时，这两方面的代价都非常大，而且我们认为这样也是不可取的。

不管是哪部分负责管理客户端缓存，ZooKeeper直接管理或ZooKeeper应用来管理，都可以通过同步或异步方式进行更新操作的客户端通知。同步方式使所有持有该状态拷贝的客户端中的状态无效，这种方式效率很低，因为客户端往往以不同的速度运行中，因此缓慢的客户端会强制其他客户端进行等待，随着客户端越来越多，这种差异就会更加频繁发生。

设计者选择的通知方式可视为一种在客户端一侧使ZooKeeper状态失效的异步方式，ZooKeeper将给客户端的通知消息队列化，这些通知会以异步的方式进行消费。这种失效方案也是可选方案，应用程序中需要解决，对于任何给定的客户端，哪些部分ZooKeeper状态需要置为无效。这种设计的选择更适合ZooKeeper的应用场景。

4.7 顺序的保障

在通过ZooKeeper实现我们的应用时，我们还要牢记一些很重要的涉及顺序性的事项。

4.7.1 写操作的顺序

ZooKeeper状态会在所有服务端所组成的全部安装中进行复制。服务端对状态变化的顺序达成一致，并使用相同的顺序执行状态的更新。例如，如果一个ZooKeeper的服务端执行了先建立一个/z节点的状态变化之后再删除/z节点的状态变化这个顺序的操作，所有的在集合中的服务端均需以相同的顺序执行这些变化。

所有服务端并不需要同时执行这些更新，而且事实上也很少这样操作。服务端更可能在不同时间执行状态变化，因为它们以不同的速度运行，即使它们运行在同种硬件下。有很多原因会导致这种时滞发生，如操作系统的调度、后台任务等。

对于应用程序来说，在不同时间点执行状态更新并不是问题，因为它们会感知到相同的更新顺序。应用程序也可能感知这一顺序，但如果ZooKeeper状态通过隐藏通道进行通信时，我们将在后续章节进行讨论。

4.7.2 读操作的顺序

ZooKeeper客户端总是会观察到相同的更新顺序，即使它们连接到不同的服务端上。但是客户端可能是在不同时间观察到了更新，如果他们还在ZooKeeper以外通信，这种差异就会更加明显。

让我们考虑以下场景：

- 客户端 c_1 更新了/z节点的数据，并收到应答。
- 客户端 c_1 通过TCP的直接连接告知客户端 c_2 ，/z节点状态发生了变化。
- 客户端 c_2 读取/z节点的状态，但是在 c_1 更新之前就观察到了这个状态。

我们称之为隐藏通道（hidden channel），因为ZooKeeper并不知道客户端之间额外的通信。现在 c_2 获得了过期数据，图4-2描述了这种情况。

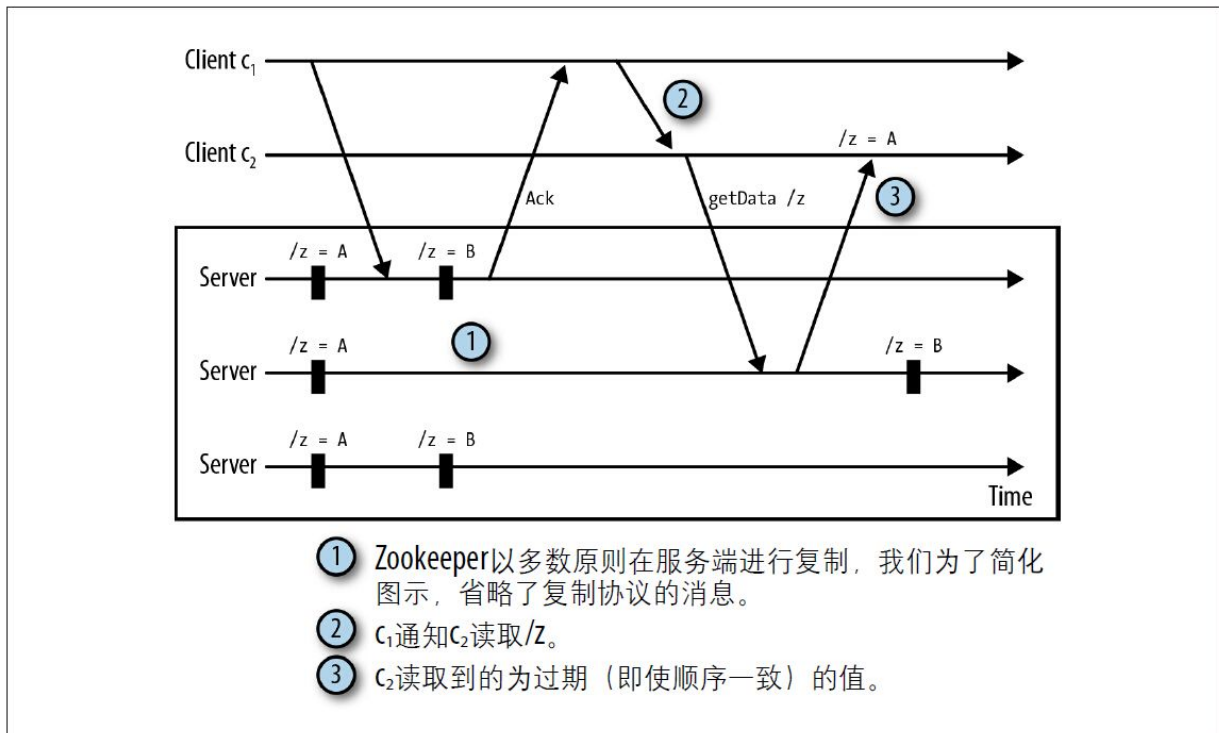


图4-2：隐藏通道问题的例子

为了避免读取到过去的的数据，我们建议应用程序使用ZooKeeper进行所有涉及ZooKeeper状态的通信。例如，为了避免刚刚描述的场景， c_2 可以在 $/z$ 节点设置监视点来代替从 c_1 直接接收消息，通过监视点， c_2 就可以知道 $/z$ 节点的变化，从而消除隐藏通道的问题。

4.7.3 通知的顺序

ZooKeeper对通知的排序涉及其他通知和异步响应，以及对系统状态更新的顺序。如ZooKeeper对两个状态更新进行排序， u 和 u' ， u' 紧随 u 之后，如果 u 和 u' 分别修改了 $/a$ 节点和 $/b$ 节点，其中客户端 c 在 $/a$ 节点设置

了监视点，c只能观察到u'的更新，即接收到u所对应通知后读取/b节点。

这种顺序可以使应用通过监视点实现安全的参数配置。假设一个znode节点/z被创建或删除表示在ZooKeeper中保存的一些配置信息变为无效的。在对这个配置进行任何实际更新之前，将创建或删除的通知发给客户端，这一保障非常重要，可以确保客户端不会读取到任何无效配置。

更具体一些，假如我们有一个znode节点/config，其子节点包含应用配置元数据：/config/m1，/config/m2，，/config/m_n。目的只是为了说明这个例子，不管这些znode节点的实际内容是什么。假如主节点应用进程通过setData更新每个znode节点，且不能让客户端只读取到部分更新，一个解决方案就是在开始更新这些配置前主节点先创建一个/config/invalid节点，其他需要读取这一状态的客户端会监视/config/invalid节点，如果该节点存在就不会读取配置状态，当该节点被删除，就意味着有一个新的有效的配置节点集合可用，客户端可以进行读取该集合的操作。

对于这个具体的例子，我们还可以使用multiop来对/config/m[1-n]这些节点原子地执行所有setData操作，而不是使用一个znode节点来标识部分修改的状态。在例子中的原子性问题，我们可以使用multiop代

替对额外znode节点或通知的依赖，不过通知机制非常通用，而且并未约束为原子性的。

因为ZooKeeper根据触发通知的状态更新对通知消息进行排序，客户端就可以通过这些通知感知到真正的状态变化的顺序。

注意：活性与安全性

在本章中，因活性广泛使用了通知机制。活性（liveness）会确保系统最终取得进展。新任务和新的从节点的通知只是关于活性的事件的例子。如果主节点没有对新任务进行通知，这个任务就永远不会被执行，至少从提交任务的客户端的视角来看，已提交的任务没有执行会导致活性缺失。

原子更新一组配置节点的例子中，情况不太一样：这个例子涉及安全性，而不是活性。在更新中读取znode节点可能会导致客户端到非一致性配置信息，而invalid节点可以确保只有当合法配置信息有效时，客户端才读取正确状态。

在我们看到的关于活性的例子中，通知的传送顺序并不是特别重要，只要最终客户端最终获知这些事件就可以继续取得进展。不过为了安全性，不按顺序接收通知也许会导致不正确的行为。

4.8 监视点的羊群效应和可扩展性

有一个问题需要注意，当变化发生时，ZooKeeper会触发一个特定的znode节点的变化导致的所有监视点的集合。如果有1000个客户端通过exists操作监视这个znode节点，那么当znode节点创建后就会发送1000个通知，因而被监视的znode节点的一个变化会产生一个尖峰的通知，该尖峰可能带来影响，例如，在尖峰时刻提交的操作延迟。可能的话，我们建议在使用ZooKeeper时，避免在一个特定节点设置大量的监视点，最好是每次在特定的znode节点上，只有少量的客户端设置监视点，理想情况下最多只设置一个。

解决该方法并不适用于所有的情况，但在以下情况下可能很有用。假设有n个客户端争相获取一个锁（例如，主节点锁）。为了获取锁，一个进程试着创建/lock节点，如果znode节点存在了，客户端就会监视这个znode节点的删除事件。当/lock被删除时，所有监视/lock节点的客户端收到通知。另一个不同的方法，让客户端创建一个有序节点/lock/lock-，回忆之前讨论的有序节点，ZooKeeper在这个znode节点上自动添加一个序列号，成为/lock/lock-xxx，其中xxx为序列号。我们可以使用这个序列号来确定哪个客户端获得锁，通过判断/lock下的所有创建的子节点的最小序列号。在该方案中，客户端通过/getChildren方法来获取所有/lock下的子节点，并判断自己创建的节

点是否是最小的序列号。如果客户端创建的节点不是最小序列号，就根据序列号确定序列，并在前一个节点上设置监视点。例如：假设我们有三个节点：`/lock/lock-001`、`/lock/lock-002`和`/lock/lock-003`，在这个例子中情况如下：

- 创建`/lock/lock-001`的客户端获得锁。
- 创建`/lock/lock-002`的客户端监视`/lock/lock-001`节点。
- 创建`/lock/lock-003`的客户端监视`/lock/lock-002`节点。

这样，每个节点上设置的监视点只有最多一个客户端。

另一方面需要注意的问题，就是当服务端一侧通过监视点产生的状态变化。设置一个监视点需要在服务端创建一个**Watcher**对象，根据YourKit (<http://www.yourkit.com/>) 的分析工具所分析，设置一个监视点会使服务端的监视点管理器的内存消耗上增加大约250到300个字节，设置非常多的监视点意味着监视点管理器会消耗大量的服务器内存。例如，如果存在一百万个监视点，估计会消耗0.3GB的内存，因此，开发者必须时刻注意设置的监视点数量。

4.9 小结

在分布式系统中，存在很多种触发一些动作的事件。ZooKeeper提供了跟踪重要事件的有效机制，使系统中的进程可以根据事件进行相应的处理。如我们之前所讨论的正常流程的应用（如，任务的执行）或崩溃故障的处理（如主节点崩溃）。

我们使用到的ZooKeeper的一个重要功能便是通知（notifications）。ZooKeeper客户端通过ZooKeeper来注册监视点，在ZooKeeper状态变化发生时，接收通知。通知的传送顺序很重要，客户端不可以自己通过某些方式观察ZooKeeper状态变化的顺序。

在同时处理多个变化的调用时，一个很有用的功能便是multi方法。这个功能使我们可以一次执行多个操作，以便在客户端对事件进行响应并改变ZooKeeper的状态时，避免在分布式应用的竞态条件。

我们希望大多数应用采用我们之前所讨论的模式，即使是该模式的各种变体也可以。我们专注于异步API，并且建议开发者也使用异步的方式，异步API可以让应用程序更有效地使用ZooKeeper资源，同时获得更高的性能。

第5章 故障处理

如果故障永远不发生，那么生活将变得更加简单。当然，没有了故障，对ZooKeeper的很多需求也就不存在了。为了有效使用ZooKeeper，我们就需要了解发生的故障的种类，以及如何处理这些故障。

故障发生的主要点有三个：ZooKeeper服务、网络、应用程序。故障恢复取决于所找到的故障发生的具体位置，不过查找具体位置并不是简单的事情。

如图5-1所展示的简单结构，只有两个进程组成应用程序，三个服务器组成了ZooKeeper的服务。进程会随机连接到其中一个服务器，也可能断开后再次连接到另一个不同的服务器，服务器使用内部协议来保持客户端之间状态的同步，对客户端呈现一致性视图。

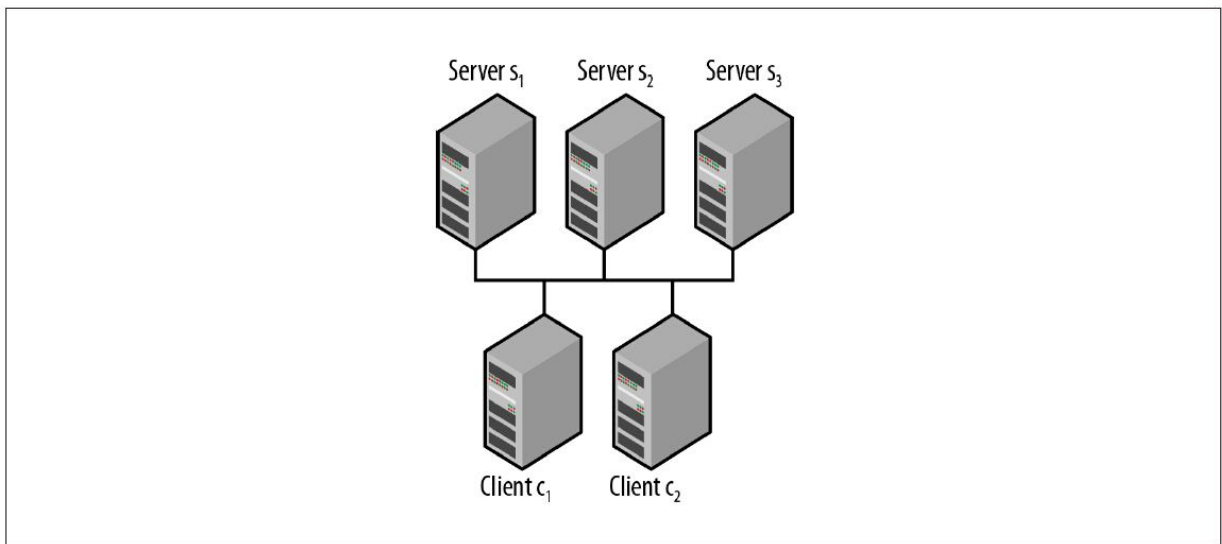


图5-1：简单的分布式应用示意图

图5-2展示了系统的不同组件中可能发生的一些故障。我们更关心如何区分一个应用中不同类型的故障，例如，如果发生网络故障， c_1 如何区分网络故障和ZooKeeper服务终端之间的区别？如果ZooKeeper服务中只有 s_1 服务器停止运行，其他的ZooKeeper服务器还会继续运行，如果此时没有网络问题， c_1 可以连接到其他的服务器上。不过，如果 c_1 无法连接到任何服务器，可能是因为当前服务不可用（也许因为集合中大多数服务器停止运行），或因为网络故障导致。

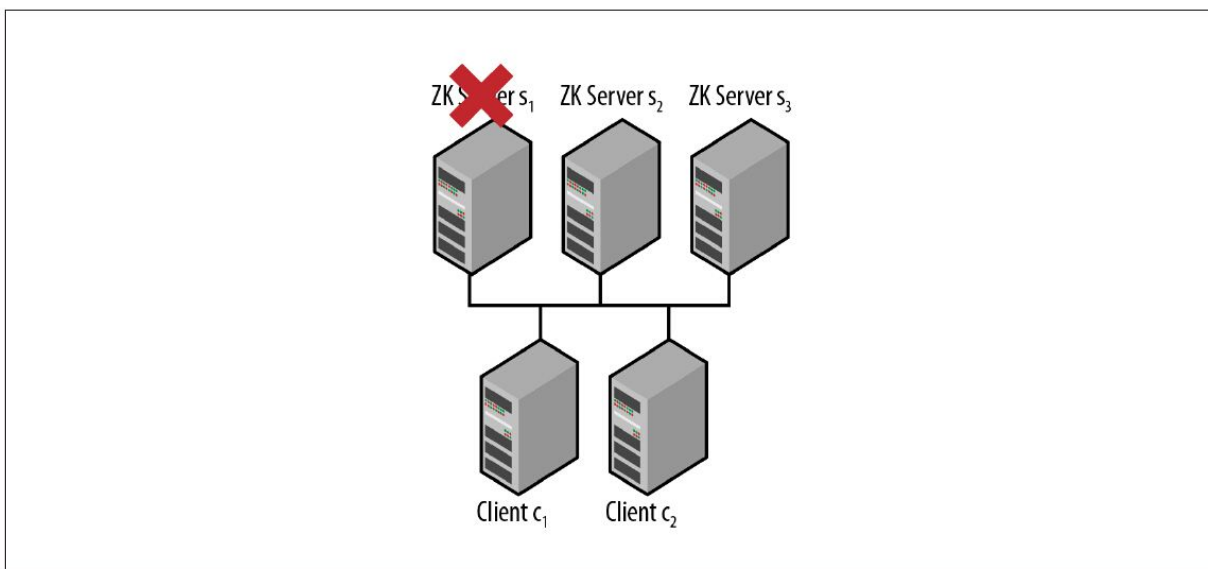


图5-2：简单的分布式系统的故障情况

这个例子展示了并不是所有的基于组件中发生的故障都可以被处理，因此ZooKeeper需要呈现系统视图，同时开发者也基于该视图进行开发。

我们再从 c_2 的视角看看图5-2，我们看到网络故障持续足够长的时间将会导致 c_1 与ZooKeeper之间的会话过期，然而即使 c_1 实际上仍然活着，ZooKeeper还是会因为 c_1 无法与任何服务器通信而声明 c_1 已经为不活动状态。如果 c_1 正在监视自己创建的临时性节点，就可以收到 c_1 终止的通知，因此 c_2 也将确认 c_1 已经终止，因为ZooKeeper也是如此通知的，即使在这个场景中 c_1 还活着。

在这个场景中， c_1 无法与ZooKeeper服务进行通信，它自己知道自己活着，但不无法确定ZooKeeper是否声明它的状态是否为终止状态，

因此必须以最坏的情况进行假设。如果 c_1 进行其他操作，但已经中止的进程不应该进行其他操作（例如改变外部资源），这样可能会破坏整个系统。如果 c_1 再也无法重新连接到ZooKeeper并发现它的会话已经不再处于活动状态，它需要确保整个系统的其他部分的一致性，并中止或执行重启逻辑，以新的进程实例的方式再次连接。

注意：事后评测ZooKeeper方案

事后评测ZooKeeper的方案的确很诱人。之前也曾经这么做过，遗憾的是，首先问题具有不确定性，如我们本章之前所介绍的。如果是ZooKeeper发生错误，事后评测也许的确猜测正确了，但如果ZooKeeper正常工作，也许猜测结果是不正确的。ZooKeeper被指定为真实性的源头，系统的设计会更简单，故障也更容易识别和诊断。

5.1 可恢复的故障

ZooKeeper呈现给使用某些状态的所有客户端进程一致性的状态视图。当一个客户端从ZooKeeper获得响应时，客户端可以非常肯定这个响应信息与其他响应信息或其他客户端所接收的响应均保持一致性。有时，ZooKeeper客户端库与ZooKeeper服务的连接会丢失，而且无法提供一致性保障的信息，当客户端库发现自己处于这种情况时，就会使用Disconnected事件和ConnectionLossException异常来表示自己无法了解当前的系统状态。

当然，ZooKeeper客户端库会积极地尝试，使自己离开这种情况，它会不断尝试重新连接另一个ZooKeeper服务器，直到最终重新建立了会话。一旦会话重新建立，ZooKeeper会产生一个SyncConnected事件，并开始处理请求。ZooKeeper还会注册之前已经注册过的监视点，并会对失去连接这段时间发生的变更产生监视点事件。

Disconnected事件和ConnectionLossException异常的产生的一个典型原因是因为ZooKeeper服务器故障。图5-3展示了这种故障的一个示例。在该例子中，客户端连接到服务器 s_2 ，其中 s_2 是两个活动ZooKeeper服务器中的一个，当 s_2 发生故障，客户端的Watcher对象就会收到Disconnected事件，并且，所有进行中的请求都会返回ConnectionLossException异常。整个ZooKeeper服务本身依然正常，因

为大多数的服务器仍然处于活动状态，所以客户端会快速与新的服务器重新建立会话。

如果客户端没有进行中的请求，这种情况只会对客户端产生很小的影响。紧随Disconnected事件之后为SyncConnected事件，客户端并不会注意到变化，但是，如果存在进行中的请求，连接丢失就会产生很大的影响。

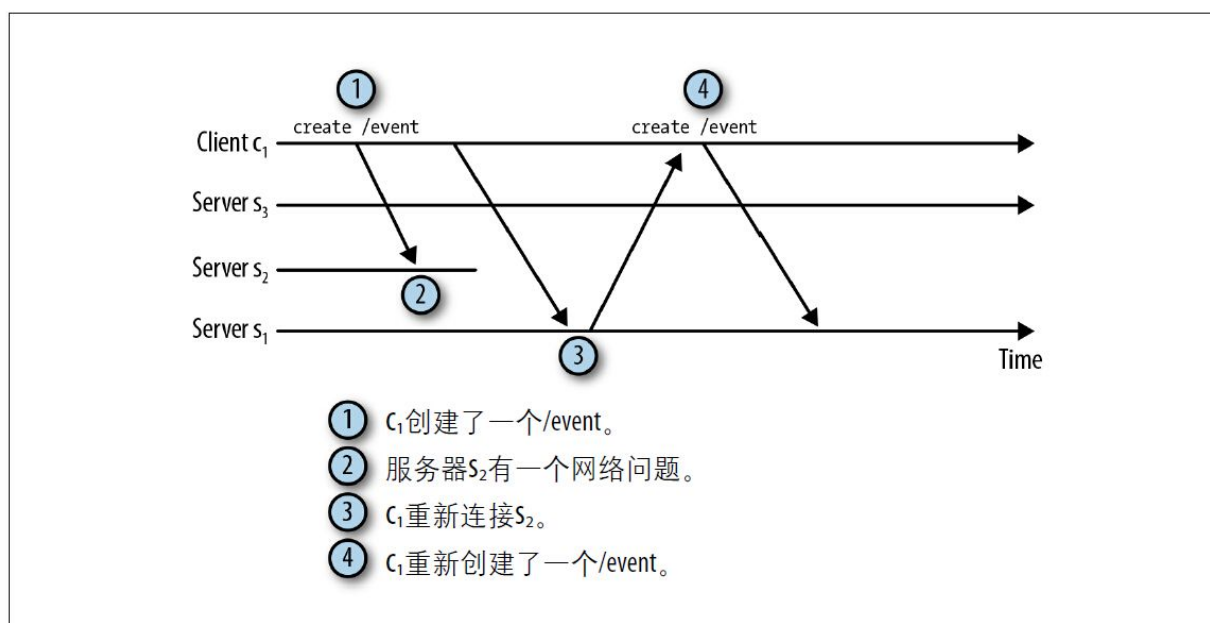


图5-3: 连接丢失的例子

如果此时客户端正在进行某些请求，比如刚刚提交了一个`create`操作的请求，当连接丢失发生时，对于同步请求，客户端会得到`ConnectionLossException`异常，对于异步请求，会得到`CONNECTIONLOSS`返回码。然而，客户端无法通过这些异常或返回

码来判断请求是否已经被处理，如我们所看到的，处理连接丢失会使我们的代码更加复杂，因为应用程序代码必须判断请求是否已经完成。处理连接丢失这种复杂情况，一个非常糟糕的方法是简单处理，当接收到`ConnectionLossException`异常或`CONNECTIONLOSS`返回码时，客户端停止所有工作，并重新启动，虽然这样可以使代码更加简单，但是，本可能是一个小影响，却变为重要的系统事件。

为了说明上面情况的原因，让我们看一个以90个客户端进程连接到一个由3个服务器组成的ZooKeeper集群的系统。如果应用采用了简单却糟糕的方式，现在有一个ZooKeeper服务器发生故障，30个客户端进程将会关闭，然后重启与ZooKeeper的会话，更糟糕的是，客户端进程在还没有与ZooKeeper连接时就关闭了会话，因此这些会话无法被显式地关闭，ZooKeeper也只能通过会话超时来监测故障。最后结果是三分之一的应用进程重启，重启却被延迟，因为新的进程必须等待之前旧的会话过期后才可以获得锁。换句话说，如果应用正确地处理连接丢失，这种情况只会产生很小的系统损坏。

开发者必须知道，当一个进程失去连接后就无法收到ZooKeeper的更新通知，尽管这听起来很没什么，但是一个进程也许会在会话丢失时错过了某些重要的状态变化。图5-4展示了这种情况的例子。客户端 c_1 作为群首，在 t_2 时刻失去了连接，但是并没发现这个情况，直到 t_4 时刻才声明为终止状态，同时，会话在 t_2 时刻过期，在 t_3 时刻另一个进程

成为群首，从 t_2 到 t_4 时刻旧的群首并不知道它自己被声明为终止状态，而另一个群首已经接管控制。

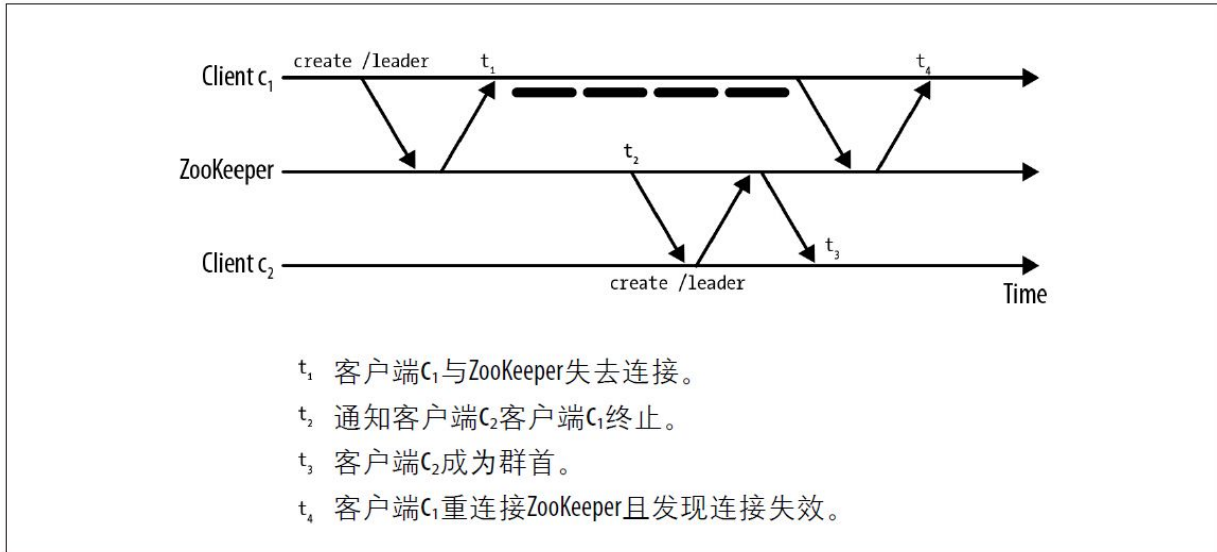


图5-4: 活动死节点的影响

如果开发者不仔细处理，旧的群首会继续担当群首，并且其操作可能与新的群首相冲突。因此，当一个进程接收到**Disconnected**事件时，在重新连接之前，进程需要挂起群首的操作。正常情况下，重新连接会很快发生，如果客户端失去连接持续了一段时间，进程也许会选择关闭会话，当然，如果客户端失去连接，关闭会话也不会使 ZooKeeper更快地关闭会话，ZooKeeper服务依然会等待会话过期时间过去以后才声明会话已过期。

注意：很长的延时与过期

当连接丢失发生时，一般情况都会快速地重新连接到另一个服务器，但是网络中断持续了一段时间可能会导致客户端重新连接ZooKeeper服务的一个长延时。一些开发者想要知道为什么ZooKeeper客户端库没有在某些时刻（比如两倍的会话超时时间）做出判断，够了，自己关闭会话吧。

对这个问题有两个答案。首先，ZooKeeper将这种策略问题的决策权交给开发者，开发者可以很容易地实现关闭句柄这种策略。其次，当整个ZooKeeper集合停机时，时间冻结，然而当整个集合恢复了，会话的超时时间被重置，如果使用ZooKeeper的进程挂起在那里，它们会发现长时间超时是因为ZooKeeper长时间的故障，ZooKeeper恢复后，客户端回到之前的正确状态，进程也就不需要额外地重启延迟时间。

已存在的监视点与Disconnected事件

为了使连接断开与重现建立会话之间更加平滑，ZooKeeper客户端库会在新的服务器上重新建立所有已经存在的监视点。当客户端连接ZooKeeper的服务器，客户端会发送监视点列表和最后已知的zxid（最终状态的时间戳），服务器会接受这些监视点并检查znode节点的修改时间戳与这些监视点是否对应，如果任何已经监视的znode节点的修改时间戳晚于最后已知的zxid，服务器就会触发这个监视点。

每个ZooKeeper操作都完全符合该逻辑，除了exists。exists操作与其他操作不同，因为这个操作可以在一个不存在的节点上设置监视点，如果我们仔细看前一段中所说的注册监视点逻辑，我们会发现存在一种错过监视点事件的特殊情况。

图5-5说明了这种特殊情况，导致我们错过了一个设置了监视点的znode节点的创建事件，客户端监视/event节点的创建事件，然而就在/event被另一个客户端创建时，设置了监视点的客户端与ZooKeeper间失去连接，在这段时间，其他客户端删除了/event，因此当设置了监视点的客户端重新与ZooKeeper建立连接并注册监视点，ZooKeeper服务器已经不存在/event节点了，因此，当处理已经注册的监视点并判断/event的监视时，发现没有/event这个节点，所以就只是注册了这个监视点，最终导致客户端错过了/event的创建事件。因为这种特殊情况，你需要尽量避免监视一个znode节点的创建事件，如果一定要监视创建事件，应尽量监视存活期更长的znode节点，否则这种特殊情况可能会伤害你。

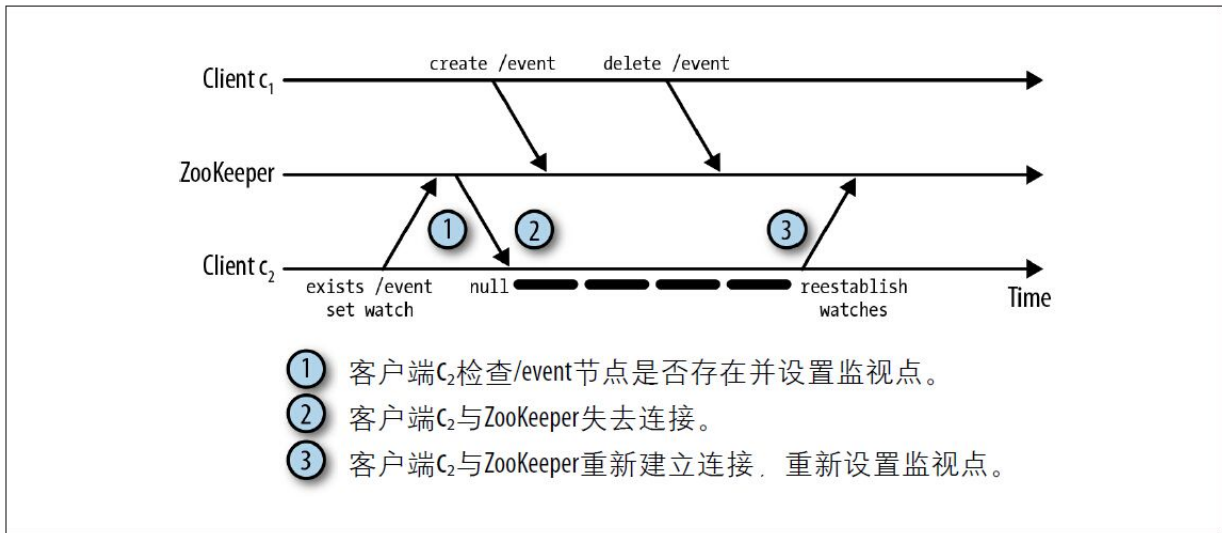


图5-5：通知的特殊情况

注意：自动重连处理危害

有些ZooKeeper的封装库通过简单的补发命令自动处理连接丢失的故障，有些情况这样做完全可以接受，但有些情况可能会导致错误的结果。例如，如果/leader节点用来建立领导权，你的程序在执行create操作建立/leader节点时连接丢失，而盲目地重试create操作会导致第二个create操作执行失败，因为/leader节点已经存在，因此该进程就会假设其他进程获得了领导权。当然，如果你知道这种情况的可能性，也了解封装库如何工作的，你可以识别并处理这种情况。有些库过于复杂，所以，如果你使用到了这种库，最好能理解ZooKeeper的原理以及该库提供给你的保障机制。

5.2 不可恢复的故障

有时，一些更糟的事情发生，导致会话无法恢复而必须被关闭。这种情况最常见的原因是会话过期，另一个原因是已认证的会话无法再次与ZooKeeper完成认证。这两种情况下，ZooKeeper都会丢弃会话的状态。

对这种状态丢失最明显的例子就是临时性节点，这种节点在会话关闭时会被删除。会话关闭时，ZooKeeper内部也会丢弃一些不可见的状态。

当客户端无法提供适当的认证信息来完成会话的认证时，或Disconnected事件后客户端重新连接到已过期的会话，就会发生不可恢复的故障。客户端库无法确定自己的会话是否已经失败，如图5-4中所看到的，直到在t4时刻，旧的客户端才已经失去连接，之后被系统其他部分声明为终止状态。

处理不可恢复故障的最简单方法就是中止进程并重启，这样可以使进程恢复原状，通过一个新的会话重新初始化自己的状态。如果该进程继续工作，首先必须要清除与旧会话关联的应用内部的进程状态信息，然后重新初始化新的状态。

注意：从不可恢复故障自动恢复的危害

简单地重新创建ZooKeeper句柄以覆盖旧的句柄，通过这种方式从不可恢复的故障中自动恢复，这听起来很吸引人。事实上，早期ZooKeeper实现就是这么做的，但是早期用户注意到这会引发一些问题。认为自己是群首的一个进程的会话中断，但是在通知其他管理线程它不是群首之前，这些线程通过新句柄操作那些只应该被群首访问的数据。为了保证句柄与会话之间一对一的对应关系，ZooKeeper现在避免了这个问题。有些情况自动恢复机制工作得很好，比如客户端只读取数据某些情况，但是如果客户端修改ZooKeeper中的数据，从会话故障中自动恢复的危害就非常重要。

5.3 群首选举和外部资源

ZooKeeper为所有客户端提供了系统的一致性视图，只要客户端与ZooKeeper进行任何交互操作（我们例子中所进行的操作），ZooKeeper都会保持同步。然而，ZooKeeper无法保护与外部设备的交互操作。这种缺乏保护的特定问题的说明，在实际环境中也经常被发现，常常发生于主机过载的情况下。

当运行客户端进程的主机发生过载，就会开始发生交换、系统颠簸或因已经超负荷的主机资源的竞争而导致的进程延迟，这些都会影响与ZooKeeper交互的及时性。一方面，ZooKeeper无法及时地与ZooKeeper服务器发送心跳信息，导致ZooKeeper的会话超时，另一方面，主机上本地线程的调度会导致不可预知的调度：一个应用线程认为会话仍然处于活动状态，并持有主节点，即使ZooKeeper线程有机会运行时才会通知会话已经超时。

图5-6通过时间轴展示了这个棘手的问题。在这个例子中，应用程序通过使用ZooKeeper来确保每次只有一个主节点可以独占访问一个外部资源，这是一个很普遍的资源中心化管理的方法，用来确保一致性。在时间轴的开始，客户端 c_1 为主节点并独占方案外部资源。事件发生顺序如下：

1.在 t_1 时刻，因为超载导致与ZooKeeper的通信停止， c_1 没有响应， c_1 已经排队等候对外部资源的更新，但是还没收到CPU时钟周期来发送这些更新。

2.在 t_2 时刻，ZooKeeper声明了 c_1 与ZooKeeper的会话已经终止，同时删除了所有与 c_1 会话关联的临时节点，包括用于成为主节点而创建的临时性节点。

3.在 t_3 时刻， c_2 成为主节点。

4.在 t_4 时刻， c_2 改变了外部资源的状态。

5.在 t_5 时刻， c_1 的负载下降，并发送已队列化的更新到外部资源上。

6.在 t_6 时刻， c_1 与ZooKeeper重现建立连接，发现其会话已经过期且丢掉了管理权。遗憾的是，破坏已经发生，在 t_5 时刻，已经在外部资源进行了更新，最后导致系统状态损坏。

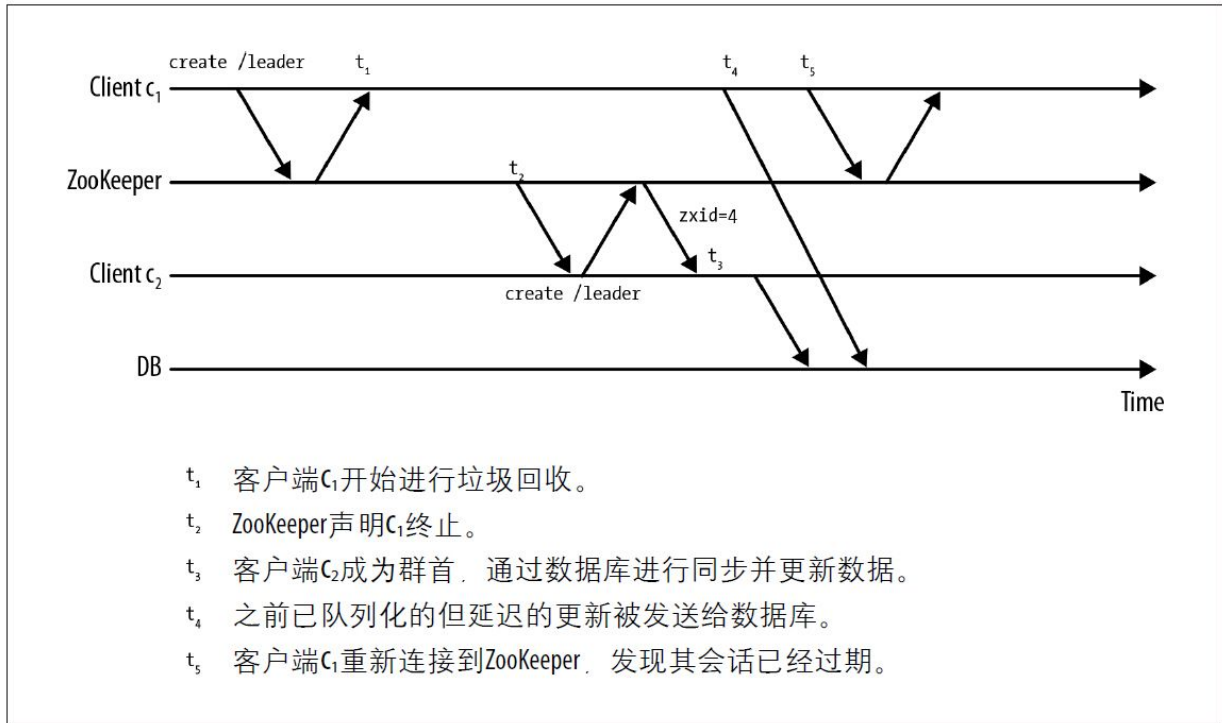


图5-6：协调外部资源

Apache HBase，作为早期采用ZooKeeper的项目，就遇到了这个问题。HBase通过区域服务器（region server）来管理一个数据库表的区域，数据被存储于分布式文件系统中，HDFS，每个区域服务器可以独占访问自己所管理的区域。HBase的每个特定的区域中，通过ZooKeeper的群首选举来确保每次只有一个区域服务器处于活动状态。

区域服务器通过Java开发，占用大量内存，当可用内存越来越少，Java会周期性地执行垃圾回收，找到释放不再使用内存，以便以后分配使用。遗憾的是，当回收大量内存时，偶尔就会出现长时间的垃圾回收周期，导致进程暂停一段时间。HBase社区发现这个时间甚至达到几十秒，就会导致ZooKeeper认为区域服务器已经终止。当垃圾回收完

成，区域服务器继续处理，有时候会先进行分布式文件系统的更新操作，这时还可以阻止数据被破坏，因为新的区域服务器接管了被认为是终止状态的区域服务器的管理权。

时钟偏移也可能导致类似的问题，在HBase环境中，因系统超载而导致时钟冻结，有时候，时钟偏移会导致时间变慢甚至落后，使得客户端认为自己还安全地处于超时周期之内，因此仍然具有管理权，尽管其会话已经被ZooKeeper置为过期。

解决这个问题有几个方法：一个方法是确保你的应用不会在超载或时钟偏移的环境中运行，小心监控系统负载可以检测到环境出现问题的可能性，良好设计的多线程应用也可以避免超载，时钟同步程序可以保证系统时钟的同步。

另一个方法是通过ZooKeeper扩展对外部设备协作的数据，使用一种名为隔离（fencing）的技巧，分布式系统中常常使用这种方法用于确保资源的独占访问。

我们用一个例子来说明如何通过隔离符号来实现一个简单的隔离。只有持有最新符号的客户端，才可以访问资源。

在我们创建代表群首的节点时，我们可以获得Stat结构的信息，其中该结构中的成员之一，**czxid**，表示创建该节点时的**zxid**，**zxid**为唯一的单调递增的序列号，因此我们可以使用**czxid**作为一个隔离的符号。

当我们对外部资源进行请求时，或我们在连接外部资源时，我们还需要提供这个隔离符号，如果外部资源已经接收到更高版本的隔离符号的请求或连接时，我们的请求或连接就会被拒绝。也就是说如果一个主节点连接到外部资源开始管理时，若旧的主节点尝试对外币资源进行某些处理，其请求将会失败，这些请求会被隔离开。即使出现系统超载或时钟偏移，隔离技巧依然可以可靠地工作。

图5-7展示了如何通过该技巧解决图5-6的情况。当 c_1 在 t_1 时刻成为群首，创建/leader节点的zxid为3（真实环境中，zxid为一个很大的数字），在连接数据库时使用创建的zxid值作为隔离符号。之后， c_1 因超载而无法响应，在 t_2 时刻，ZooKeeper声明 c_1 终止， c_2 成为新的群首。 c_2 使用4所作为隔离符号，因为其创建/leader节点的创建zxid为4。在 t_3 时刻， c_2 开始使用隔离符号对数据库进行操作请求。在 t_4 时刻， c_1 的请求到达数据库，请求会因传入的隔离符号（3）小于已知的隔离符号（4）而被拒绝，因此避免了系统的破坏。

不过，隔离方案需要修改客户端与资源之间的协议，需要在协议中添加zxid，外部资源也需要持久化保存来跟踪接收到的最新的zxid。

一些外部资源，比如文件服务器，提供局部锁来解决隔离的问题。不过这种锁也有很多限制，已经被ZooKeeper移出并声明终止状态的群首可能仍然持有一个有效的锁，因此会阻止新选举出的群首获取

这个锁而导致无法继续，在这种情况下，更实际的做法是使用资源锁来确定领导权，为了提供有用信息的目的，由群首创建/leader节点。

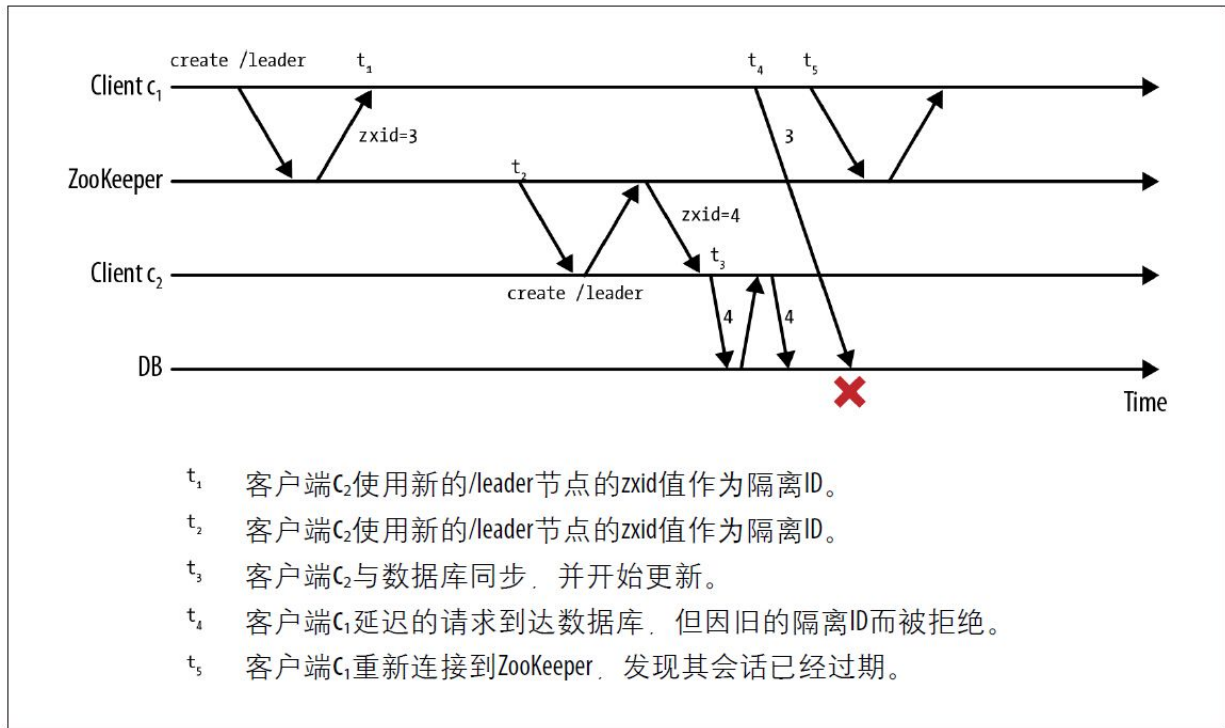


图5-7：通过ZooKeeper使用隔离

5.4 小结

在分布式系统中，故障是无法避免的事实。ZooKeeper无法使故障消失，而是提供了处理故障的一套框架。为了更有效地处理故障，开发者在使用ZooKeeper时需要处理状态变化的事件、故障代码以及ZooKeeper抛出的异常。不过，不是所有故障在任何情况下都采用一样的方式去处理，有时开发者需要考虑连接断开的状态，或处理连接断开的异常，因为进程并不知道系统其他部分发生了什么，甚至不知道自己进行中的请求是否已经执行。在连接断开这段时间，进程不能假设系统中的其他部分还在运行中。即使ZooKeeper客户端库与ZooKeeper服务器重新建立了连接，并重新建立监视点，也需要校验之前进行中的请求的结果是否成功执行。

第6章 ZooKeeper注意事项

在前面的章节中讲述了如何使用ZooKeeper进行编码开发，实现了从一些基本场景到一些复杂的情况。在本章中，我们集中讨论ZooKeeper的某些棘手的问题，主要涉及会话的语义学和顺序性的问题。本章所讲述的内容可能并不会影响你进行开发，但当你遇到本章所讲述的某些问题时，你会发现这些内容对你很有好处。

本章与其他章节的结构不同，我们并没有采用线性方式进行描述，而是通过一个个混合性的问题逐一展开。每一小节的内容都是独立的内容，所以你可以单独阅读某一节的内容。在之前的章节中，我们已经讨论了我们遇到的一些棘手问题，但在这一章中，虽然没有必要再讨论这些已经在其他地方讨论过的问题，但是这些问题仍然很重要，因为许多开发人员都曾遇到这些问题。

6.1 使用ACL

正常情况下，你希望在管理或实施章节看到访问控制的内容，然而，对于ZooKeeper，开发人员往往负责管理访问控制的权限，而不是管理员。这是因为每次创建znode节点时，必须设置访问权限，而且子节点并不会继承父节点的访问权限。访问权限的检查也是基于每一个znode节点的，如果一个客户端可以访问一个znode节点，即使这个客户端无权访问该节点的父节点，仍然可以访问这个znode节点。

ZooKeeper通过访问控制表（ACL）来控制访问权限。一个ACL包括以下形式的记录：scheme: auth-info，其中scheme对应了一组内置的鉴权模式，auth-info为对于特定模式所对应的方式进行编码的鉴权信息。ZooKeeper通过检查客户端进程访问每个节点时提交上来的授权信息来保证安全性。如果一个进程没有提供鉴权信息，或者鉴权信息与要请求的znode节点的信息不匹配，进程就会收到一个权限错误。

为了给一个ZooKeeper增加鉴权信息，需要调用addAuthInfo方法，形式如下：

```
void addAuthInfo(  
    String scheme,  
    byte auth[]  
)
```

其中：

`scheme`

表示所采用的鉴权模式。

`auth`

表示发送给服务器的鉴权信息。该参数的类型为`byte[]`类型，不过大部分的鉴权模式需要一个`String`类型的信息，所以你可以通过`String.getBytes ()`来将`String`转换为`byte[]`。

一个进程可以在任何时候调用`addAuthInfo`来添加鉴权信息。一般情况下，在`ZooKeeper`句柄创建后就会调用该方法来添加鉴权信息。进程中可以多次调用该方法，为一个`ZooKeeper`句柄添加多个权限的身份。

6.1.1 内置的鉴权模式

`ZooKeeper`提供了4种内置模式进行ACL的处理。其中一个我们之前已经使用过，通过`OPEN_ACL_UNSAFE`常量隐式传递了ACL策略，这种ACL使用`world`作为鉴权模式，使用`anyone`作为`auth-info`，对于`world`这种鉴权模式，只能使用`anyone`这种`auth-info`。

另一种特殊的内置模式为管理员所使用的super模式，该模式不会被列入到任何ACL中，但可以用于ZooKeeper的鉴权。一个客户端通过super鉴权模式连接到ZooKeeper后，不会被任何节点的ACL所限制。关于super方案的更多内容，请参考10.1.5节。

我们通过下面的例子来介绍另外两个鉴权模式。

当ZooKeeper以一个空树开始，只有一个znode节点：/，这个节点对所有人开放，我们假设管理员Amy负责配置ZooKeeper服务，Amy创建/apps节点，用于所有使用服务的应用需要创建节点的父节点，她现在需要锁定服务，所以她设置为/和/apps节点设置的ACL为：

```
digest:amy:Iq0onHjzb4KyxPAP8YWOIC8zzwY=, READ | WRITE | CREATE | DELETE | ADMIN
```

该ACL只有一条记录，为Amy提供所有访问权限。Amy使用amy作为用户ID信息。

digest为内置鉴权模式，该模式的auth-info格式为userid: passwd_digest，当调用addAuthInfo时需要设置ACL和userid: password信息。其中passwd_digest为用户密码的加密摘要。在这个ACL例子中，Iq0onHjzb4KyxPAP8YWOIC8zzwY=为passwd_digest，因此当Amy调用addAuthInfo方法，auth参数传入的为amy: secret字符串的字节数

组，Amy使用下面的DigestAuthenticationProvider来为她的账户amy生成摘要信息。

```
java -cp $ZK_CLASSPATH \
    org.apache.zookeeper.server.auth.DigestAuthenticationProvider amy:secret
....
amy:secret->amy:Iq0onHjzb4KyxPAp8YW0IC8zzwY=
```

amy: 后面生成的字符串为密码摘要信息，也就是我们在ACL记录总使用的信息。当Amy需要向ZooKeeper提供鉴权信息时，她就要使用digest amy: secret。例如，当Amy使用zkCli.sh连接到ZooKeeper，她可以通过以下方式提供鉴权信息：

```
[zk: localhost:2181(CONNECTED) 1] addauth digest amy:secret
```

为了避免在后面的例子中写出所有的摘要信息，我们将使用XXXXXX作为占位符来简单的表示摘要信息。

Amy想要设置一个子树，用于一个名为SuperApp的应用，该应用由开发人员Dom所开发，因此她创建了/apps/SuperApp节点，设置ACL如下：

```
digest:dom:XXXXX, READ | WRITE | CREATE | DELETE | ADMIN
digest:amy:XXXXX, READ | WRITE | CREATE | DELETE | ADMIN
```

该ACL由两条记录组成，一个由Dom使用，一个由Amy使用。这些记录对所有以dom或amy密码信息认证的客户端提供了全部权限。

注意，根据ACL中的Dom的记录，他对/apps/SuperApp节点具有ADMIN权限，他有权限修改ACL，这就意味着Dom可以删除Amy访问/apps/SuperApp节点的权限。当然Amy具有super的访问权限，所以 she 可以随时访问任何znode节点，即使Dom删除了她的访问权限。

Dom使用ZooKeeper来保存其应用的配置信息，因此他创建了/apps/SuperApp/config节点来保存配置信息。之后他使用我们在之前例子中介绍的模式OPEN_ACL_UNSAFE来创建znode节点，因为Dom认为/apps和/apps/SuperApp的访问是受限制的，所以也能保护/apps/SuperApp/config节点的访问。我们后面就会看到，这样做称为UNSAFE。

我们假设一个名为Gabe的人具有ZooKeeper服务的网络访问权限。因为ACL的策略设置，Gabe无法访问/app或/apps/SuperApp节点，Gabe也无法获取/apps/SuperApp节点的子节点列表。但是，也许Gabe猜测Dom使用ZooKeeper保存配置信息，config这个名字对于配置文件信息也非常显而易见，因此他连接到ZooKeeper服务，调用getData方法获取/apps/SuperApp/config节点的信息。因为该znode节点采用了开放的ACL策略，Gabe可以获取该节点信息。还不止这些，Gabe可以修改、删除该节点，甚至限制/apps/SuperApp/config节点的访问权限。

假设Dom意识到这个问题，修改/apps/SuperApp/config节点的ACL策略为：

```
digest:dom:XXXXX, READ | WRITE | CREATE | DELETE | ADMIN
```

随着事情的发展，Dom得到一个新的开发人员Nico的帮助，来一同完善SuperApp。Nico需要访问SuperApp的子树，因此Dom修改了子树的ACL策略，将Nico添加进来。新的ACL策略为：

```
digest:dom:XXXXX, READ | WRITE | CREATE | DELETE | ADMIN  
digest:nico:XXXXX, READ | WRITE | CREATE | DELETE | ADMIN
```

注意：用户名和密码的摘要信息从何而来？

你也许注意到我们用于摘要的用户名和密码似乎凭空而来。实际上确实如此。这些用户名或密码不用对应任何真实系统的标识，甚至用户名也可以重复。也许有另一个开发人员叫Amy，并且开始和Dom和Nico一同工作，Dom可以使用amy: XXXXX来添加她的ACL策略，只是在这两个Amy的密码一样时会发生冲突，因为这样就导致她们俩可以互相访问对方的信息。

现在Dom和Nico具有了他们需要完成的SuperApp的所需的访问权限。应用部署到生产环境，然而Dom和Nico并不想提供进程访问ZooKeeper数据时所使用的密码信息，因此他们决定通过SuperApp所运行的服务器的网络地址来限制数据的访问权限。例如所有10.11.12.0/24网络中服务器，因此他们修改了SuperApp子树的znode节点的ACL为：

```
digest:dom:XXXXX, READ | WRITE | CREATE | DELETE | ADMIN
digest:nico:XXXXX, READ | WRITE | CREATE | DELETE | ADMIN
ip:10.11.12.0/24, READ
```

ip鉴权模式需要提供网络的地址和掩码，因为需要通过客户端的地址来进行ACL策略的检查，客户端在使用ip模式的ACL策略访问znode节点时，不需要调用addAuthInfo方法。

现在，任何在10.11.12.0/24网段中运行的ZooKeeper客户端都具有SuperApp子树的znode节点的读取权限。该鉴权模式假设IP地址无法被伪造，这个假设也许并不能适合于所有环境中。

6.1.2 SASL和Kerberos

前一节中的例子还有几个问题。首先，如果新的开发人员加入或离开组，管理员就需要改变所有的ACL策略，如果我们通过组来避免这种情况，那么事情就会好一些。其次，如果我们想修改某开发人员的密码，我们也需要修改所有的ACL策略。最后，如果网络不可信，无论是digest模式还是ip模式都不是最合适的模式。我们可以通过使用ZooKeeper提供的sasl模式来解决这些问题。

SASL表示简单认证与安全层（Simple Authentication and Security Layer）。SASL将底层系统的鉴权模型抽象为一个框架，因此应用程序可以使用SASL框架，并使用SASL支持多各种协议。在ZooKeeper

中，SASL常常使用Kerberos协议，该鉴权协议提供之前我们提到的那些缺失的功能。在使用SASL模式时，使用sasl作为模式名，id则使用客户端的Kerberos的ID。

SASL是ZooKeeper的扩展鉴权模式，因此，需要通过配置参数或Java系统中参数激活该模式。如果你采用ZooKeeper的配置文件方式，需要使用authProvider.XXX配置参数，如果你想要通过系统参数方式，需要使用zookeeper.authProvider.XXX作为参数名。这两种情况下，XXX可以为任意值，只要没有任何重名的authProvider，一般XXX采用以0开始的一个数字。配置项的参数值为org.apache.zookeeper.server.auth.SASLAuthenticationProvider，这样就可以激活SASL模式。

6.1.3 增加新鉴权模式

ZooKeeper中还可以使用其他的任何鉴权模式。对于激活新的鉴权模式来说只是简单的编码问题。在org.apache.zookeeper.server.auth包中提供了一个名为AuthenticationProvider的接口类，如果你实现你自己的鉴权模式，你可以将你的类发布到服务器的classpath下，创建zookeeper.authProvider名称前缀的Java系统参数，并将参数值设置为你实现AuthenticationProvider接口的实际的类名。

6.2 恢复会话

假如你的ZooKeeper客户端崩溃，之后恢复运行，应用程序在恢复运行后需要处理一系列问题。首先，应用程序的ZooKeeper状态还处于客户端崩溃时的状态，其他客户端进程还在继续运行，也许已经修改了ZooKeeper的状态，因此，建议客户端不要使用任何之前从ZooKeeper获取的缓存状态，而是使用ZooKeeper作为协作状态的可信来源。

例如，在我们的主从实现中，如果主要主节点崩溃并恢复，与此同时，集群也许已经对分配的任务完成了切换到备份主节点的故障转移。当主要主节点恢复后，就不能再认为自己是主节点，并认为待分配任务列表已经发生变化。

第二个重要问题是客户端崩溃时，已经提交给ZooKeeper的待处理操作也许已经完成了，由于客户端崩溃导致无法收到确认消息，ZooKeeper无法保证这些操作肯定会成功执行，因此，客户端在恢复时也许需要进行一些ZooKeeper状态的清理操作，以便完成某些未完成的任务。例如，如果我们的主节点崩溃前进行了一个已分配任务的列表删除操作，在恢复并再次成为主要主节点时，就需要再次删除该任务。

尽管到目前为止我们讨论的都是客户端崩溃的案例，本节中还需要讨论会话过期的问题。对于会话过期，不能认为是客户端崩溃。会话也许因为网络问题或其他问题过期，比如Java中的垃圾回收中断，在会话过期的情况下，客户端需要考虑ZooKeeper状态也许已经发生了改变，或者客户端对ZooKeeper的请求也许并未完成。

6.3 当znode节点重新创建时，重置版本号

这个话题似乎是显而易见的，但还是要再次强调，znode节点被删除并重建后，其版本号将会被重置。如果应用程序在一个znode节点重建后，进行版本号检查会导致错误的发生。

假设客户端获取了一个znode节点（如：/z）的数据，改变该节点的数据，并基于版本号为1的条件进行回写，如果在客户端更新该节点数据时，znode节点被删除并重建了，版本号还是会匹配，但现在也许保存的是错误的数据了。

另一种可能的情况，在一个znode节点删除中和重建中，对于znode节点发生的变化的情况。此时进行节点的更新操作，setData操作永远不会修改znode节点的数据，这种情况下，通过检查版本号并不能提供znode节点的变化情况，znode节点也许会被更新任意次，但其版本号仍然为0。

6.4 sync方法

如果应用客户端只对ZooKeeper的读写来通信，应用程序就不用考虑sync方法。sync方法的设计初衷，是因为与ZooKeeper的带外通信可能会导致某些问题，这种通信常常称为隐蔽通道（hidden channel），在4.7节中我们已经对此问题进行过解释。问题主要源于一个客户端c也许通过某些直接通道（例如，c和c'之间通过TCP连接进行通讯）来通知另一个客户端c进行ZooKeeper状态变化，但是当c读取ZooKeeper的状态时，却并未发现变化情况。

这一场景发生的原因，可能因为这个客户端所连接的服务器还没来得及处理变化情况，而sync方法可以用于处理这种情况。sync为异步调用的方法，客户端在读操作前调用该方法，假如客户端从某些直接通道收到了某个节点变化的通知，并要读取这个znode节点，客户端就可以通过sync方法，然后再调用getData方法：

```
...  
zk.sync(path, voidCb, ctx);①
```

```
zk.getData(path, watcher, dataCb, ctx);②
```

```
...
```

①sync方法接受一个path参数，一个void返回类型的回调方法的示例，一个上下文对象实例。

②getData方法与之前介绍的调用方式一样。

sync方法的path参数指示需要进行操作的路径。在系统内部，sync方法实际上并不会影响ZooKeeper，当服务端处理sync调用时，服务端会刷新群首与调用sync操作的客户端c所连接的服务端之间的通道，刷新的意思就是说在调用getData的返回数据的时候，服务端确保返回所有客户端c调用sync方法时所有可能的变化情况。在上面的隐蔽通道的情况中，变化情况的通信会先于sync操作的调用而发生，因此当c收到getData调用的响应，响应中必然会包含c所通知的变化情况。注意，在此时该节点也可能发生了其他变化，因此在调用getData时，ZooKeeper只保证所有变化情况能够返回。

使用sync还有一个注意事项，这个需要深入ZooKeeper内部的技术问题（你可以选择跳过此部分）。因为ZooKeeper的设计初衷是用于快速读取以及以读为主要负载的扩展性考虑，所以简化了sync的实现，同时与其他常规的更新操作（如create、setData或删除）不同，sync操作并不会进入执行管道之中。sync操作只是简单地传递到群首，之后群首会将响应包队列化，传递给群组成员，之后发送响应包。不过还有另外一种可能，仲裁机制确定的群首l，现在已经不被仲裁组成员所认可，仲裁组成员现在选举了另一个群首l'，在这种情况下，群首l可

能无法处理所有的更新操作的同步，而sync调用也就可能无法履行其保障。

ZooKeeper的实现中，通过以下方式处理上面的问题，ZooKeeper中的仲裁组成员在放弃一个群首时会通知该群首，通过群首与群组成员之间的tickTime来控制超时时间，当它们之间的TCP连接丢失，群组成员在收到socket的异常后就会确定群首是否已经消失。群首与群组成员之间的超时会快于TCP连接的中止，虽然的确存在这种极端情况导致错误的可能，但是在我们现有经验中还未曾遇到过。

在邮件列表的讨论组中，曾经多次讨论过该问题，希望将sync操作放入执行管道，并可以一并消除这种极端情况。就目前来看，ZooKeeper的实现依赖于合理的时序假设，因此没有什么问题。

6.5 顺序性保障

虽然ZooKeeper声明对一个会话中所有客户端操作提供顺序性的保障，但还是会存在ZooKeeper控制之外某些情况，可能会改变客户端操作的顺序。开发人员需要注意以下参考信息，以便保证程序按照你所期望的行为执行。我们将会讨论三种情况。

6.5.1 连接丢失时的顺序性

对于连接丢失事件，ZooKeeper会取消等待中的请求，对于同步方法的调用客户端库会抛出异常，对于异步请求调用，客户端调用的回调函数会返回结果码来标识连接丢失。在应用程序的连接丢失后，客户端库不会再次重新提交请求，因此就需要应用程序对已经取消的请求进行重新提交的操作。所以，在连接丢失的情况下，应用程序可以依赖客户端库来解决所有后续操作，而不能依赖ZooKeeper来承担这些操作。

为了明白连接丢失对应用程序的影响，让我们考虑以下事件顺序：

- 1.应用程序提交请求，执行Op1操作。

2.客户端检测到连接丢失，取消了Op1操作的请求。

3.客户端在会话过期前重新连接。

4.应用程序提交请求，执行Op2操作。

5.Op2执行成功。

6.Op1返回CONNECTIONLOSS事件。

7.应用程序重新提交Op1操作请求。

在这种情况下，应用程序按顺序提交了Op1和Op2请求，但Op2却先于Op1成功执行。当应用程序在Op1的回调函数中发现连接丢失情况后，应用程序再次提交请求，但是假设客户端还没有成功重连，再次提交Op1还是会得到连接丢失的返回，因此在重新连接前存在一个风险，即应用程序进入重新提交Op1请求的无限循环中，为了跳出该循环，应用程序可以设置重试的次数，或者在重新连接时间过长时关闭句柄。

在某些情况中，确保Op1先于Op2成功执行可能是重要问题。如果Op2在某种程度上依赖与Op1操作，为了避免Op2在Op1之前成功执行，我们可以等待Op1成功执行之后在提交Op2请求，这种方法我们在很多主从应用的示例代码中使用过，以此来保证请求按顺序执行。通常，等待Op1操作结果的方法很安全，但是带来了性能上的损失，因

为应用程序需要等待一个请求的操作结果再提交下一个，而不能将这些操作并行化。

注意：假如我们摆脱CONNECTIONLOSS会怎样？

CONNECTIONLOSS事件的存在的原因，因为请求正在处理中，但客户端与服务端失去了连接。比如，对于一个create操作请求，在这种情况下，客户端并不知道请求是否处理完成，然而客户端可以询问服务端来确认请求是否成功执行。服务端可以通过内存或日志中缓存的信息记录，知道自己处理了哪些请求，因此这种方式也是可行的。如果开发社区最终改变了ZooKeeper的设计，在重新连接时服务端访问这些缓存信息，我们就可以取消无法保证前置操作执行成功的限制，因为客户端可以在需要时重新执行待处理的请求。但到目前为止，开发人员还需要注意这一限制，并妥善处理连接丢失的事件。

6.5.2 同步API和多线程的顺序性

目前，多线程应用程序非常普遍，如果你在多线程环境中使用同步API，你需要特别注意顺序性问题。一个同步ZooKeeper调用会阻塞运行，直到收到响应信息，如果两个或更多线程向ZooKeeper同时提交了同步操作，这些线程中将会被阻塞，直到收到响应信息，ZooKeeper会顺序返回响应信息，但操作结果可能因线程调度等原因导致后提交

的操作而先被执行。如果ZooKeeper返回响应包与请求操作非常接近，你可能会看到以下场景。

如果不同的线程同时提交了多个操作请求，可能是一些并不存在某些直接联系的操作，或不会因任意的执行顺序而导致一致性问题的操作。但如果这些操作具有相关性，客户端应用程序在处理结果时需要注意这些操作的提交顺序。

6.5.3 同步和异步混合调用的顺序性

还有另外一种可能出现顺序混乱的情况。假如你通过异步操作提交了两个请求，Aop1和Aop2，不管这两个操作具体是什么，只是通过异步提交的两个操作。在Aop1的回调函数中，你进行了一个同步调用，Sop1，该同步调用阻塞了ZooKeeper客户端的分发线程，这样就会导致客户端应用程序接收Sop1的结果之后才能接收到Aop2的操作结果。因此应用程序观察到的操作结果顺序为Aop1、Sop1、Aop2，而实际的提交顺序并非如此。

通常，混合同步调用和异步调用并不是好方法，不过也有例外的情况，例如当启动程序时，你希望在处理之前先在ZooKeeper中初始化某些数据，虽然这时可以使用Java锁或其他某些机制处理，但采用一个或多个同步调用也可以完成这项任务。

6.6 数据字段和子节点的限制

ZooKeeper默认情况下对数据字段的传输限制为**1MB**，该限制为任何节点数据字段的最大可存储字节数，同时也限制了任何父节点可以拥有的子节点数。选择**1MB**是随意制定的，在某种意义上来说，没有任何基本原则可以组织ZooKeeper使用其他值，更大或更小的限制。然而设置限制值可以保证高性能。如果一个znode节点可以存储很大的数据，就会在处理时消耗更多的时间，甚至在处理请求时导致处理管道的停滞。如果一个客户端在一个拥有大量子节点的znode节点上执行getChildren操作，也会导致同样的问题。

ZooKeeper对数据字段的大小和子节点的数量默认的限制值已经足够大，你需要避免接近该限制值的使用。我们也可以开启更大的限制值，来满足非常大量的应用的需求，如果你有特殊用途确实需要修改该限制值，你可以通过在10.1.6节所描述的方式来改变该值。

6.7 嵌入式ZooKeeper服务器

很多开发人员考虑在其应用中嵌入ZooKeeper服务器，以此来隐藏对ZooKeeper的依赖。对于“嵌入式”，我们指在应用内部实例化ZooKeeper服务器的情况。该方法使应用的用户对ZooKeeper的使用透明化，虽然这个主意听起来很吸引人（毕竟谁都不喜欢额外的依赖），但我们还是不建议这样做。我们观察到一些采用嵌入式方式的应用中所遇到的问题，如果ZooKeeper发生错误，用户将会查看与ZooKeeper相关的日志信息，从这个角度看，对用户已经不再是透明化的，而且应用开发人员也许无法处理这些ZooKeeper的问题。甚至更糟的是，整个应用的可用性和ZooKeeper的可用性被耦合在一起，如果其中一个退出，另一个也必然会退出。ZooKeeper常常被用来提供高可用服务，但对于应用中嵌入ZooKeeper的方式却降低了其最强的优势。

虽然我们不建议采用嵌入式ZooKeeper服务器，但也没有什么理论阻止一个人这样做，例如，在ZooKeeper测试程序中，因此，如果你真的想要采用这种方式，ZooKeeper的测试程序是一个很好的资源，教你如何做。

6.8 小结

有时，使用ZooKeeper进行开发需要非常小心，我们将读者的注意力集中到顺序性的保障和会话的语义学之上，起初这些概念看起来很容易理解，从某种意义上来说，的确是这样，但是在某些重要的极端情况下，开发人员需要小心应对。本章的主要目的就是提供一些开发的指导方针，并涉及某些极端情况的说明。

第7章 C语言客户端

虽然ZooKeeper的Java版本接口是主要被使用的一个接口，但C语言版本的ZooKeeper客户端绑定也同样受ZooKeeper开发人员所欢迎，并以此为基础形成了其他语言的绑定接口。本章将集中介绍该绑定接口，介绍如何使用C语言API来进行ZooKeeper应用的开发。我们将会通过C语言再次实现主从例子中的主节点示例，通过该示例来展示与Java语言的API的不同之处。

对于C语言的API，主要参考为ZooKeeper发行包中的zookeeper.h文件，以及在项目发行包中的README文件中所介绍的构建客户端库的操作步骤。或者，你可以采用ant compile-native命令，这样就可以自动生成这些。在开始进行编码之前，我们需要先介绍一下如何设置开发环境，来帮助大家快速开始。

当我们构建C客户端，将会生成两个库，一个用于多线程客户端的库，另一个为用于单线程的客户端库。本章中大多数示例假设采用多线程客户端库，直到本章结尾我们再讨论单线程版本的客户端库。我们建议读者更关注多线程的实现方式。

7.1 配置开发环境

在ZooKeeper的发行包中，已经包含了在任何平台上运行的JAR包文件。为了使用C语言在本地进行编译，在编译我们的C版本ZooKeeper应用之前，我们需要先构建必需的共享库。幸好，ZooKeeper提供了简单的方式来构建这些库。

构建ZooKeeper本地库的最简单的方式是使用ant构建工具。在你解压缩的ZooKeeper发行包的目录中，有一个名为build.xml的文件，该文件包含了ant构建所需的构建步骤。你还需要用到automake、autoconf和cppunit这些工具，如果你使用Linux操作系统，需要确保这些工具在你的主机中可用。在Windows中Cygwin已经提供了这些工具。在Mac OS X系统中，你可以使用开源包管理工具，如Fink、Brew或MacPorts。

一旦安装了所有必需的工具，你可以采用以下方式构建ZooKeeper的库：

```
ant compile-native
```

当构建完成，你可以在build/c/build/usr/lib中发现链接库文件，在build/c/build/usr/include/zookeeper发现你所需要的头文件。

7.2 开始会话

与ZooKeeper进行任何操作之前，我们首先需要有一个`zhandle_t`句柄。我们通过调用`zookeeper_init`函数来获取句柄，函数定义如下：

```
ZOOAPI zhandle_t *zookeeper_init(const char *host, ①  
  
                                watcher_fn fn, ②  
  
                                int rcv_timeout, ③  
  
                                const clientid_t *clientid, ④  
  
                                void *context, ⑤  
  
                                int flags); ⑥
```

①包含ZooKeeper服务集群的主机地址的字符串，地址格式为
`host: port`，每组地址以逗号分隔。

②用于处理事件的监视点函数，将在下一节介绍其定义。

③会话过期时间，以毫秒为单位。

④之前已建立的一个会话的客户端ID，用于客户端重新连接。在建立会话时，通过调用`zoo_client_id`来获取客户端ID。指定参数0来开始新的会话。

⑤返回的`zkhandle_t`句柄所使用的上下文对象。

⑥该参数暂时没有使用，因此设置为0即可。

注意：关于ZOOAPI宏的定义

ZOOAPI宏用于在Windows系统中构建ZooKeeper库。ZOOAPI宏的可能值有：`__declspec (dllexport)`、`__declspec (dllimport)`和空。对于`__declspec (dllexport)`和`__declspec (dllimport)`关键词，将会分别导出或导入符号到DLL文件中。如果你不在Windows中构建树，保持ZOOAPI为空。当然，原则上在Windows中构建树，你不需要进行任何配置，发行包中的配置已经处理了。

`zookeeper_init`调用也许在实际完成会话建立前返回，因此只有当收到ZOO_CONNECTED_STATE事件时才可以认为会话建立完成。该事件可以通过监视点函数的实现来处理，该函数的定义如下：

```
typedef void (*watcher_fn)(zhandle_t *zh, ①  
  
    int type, ②  
  
    int state, ③  
  
    const char *path, ④  
  
    void *watcherCtx); ⑤
```

①观察点函数引用的ZooKeeper句柄。

②事件类型：ZOO_CREATED_EVENT、
ZOO_DELETED_EVENT、ZOO_CHANGED_EVENT、
ZOO_CHILD_EVENT、ZOO_SESSION_EVENT。

③连接状态。

④被观察并触发事件的znode节点路径，如果事件为会话事件，路径为null。

⑤观察点的上下文对象。

以下为一个监视点函数的实现示例：

```
static int connected = 0;
static int expired = 0;
void main_watcher (zhandle_t *zkh,
                  int type,
                  int state,
                  const char *path,
                  void* context)
{
    if (type == ZOO_SESSION_EVENT) {
        if (state == ZOO_CONNECTED_STATE) {
            connected = 1;①

            } else if (state == ZOO_NOTCONNECTED_STATE ) {
                connected = 0;
            } else if (state == ZOO_EXPIRED_SESSION_STATE) {
                expired = 1;②

                connected = 0;
                zookeeper_close(zkh);
            }
        }
    }
```

①在接收到ZOO_CONNECTED_STATE事件后设置状态为已连接状态。

②在接收到ZOO_EXPIRED_SESSION_STATE事件后设置为过期状态（并关闭会话句柄）。

注意：监视点数据结构

ZooKeeper中，只要设置了监视点，就无法移除监视点，因此不管之后进程是否关心会话中的监视点，都需要保留监视点数据结构的对象，因为最后还可能调用该函数。在Java中不需要关心这个问题，因为Java中采用自动垃圾回收机制。

将所有操作串在一起，我们的主节点的init函数如下所示：

```
static int server_id;
int init (char* hostPort) {
    srand(time(NULL));
    server_id = rand();①

    zoo_set_debug_level(ZOO_LOG_LEVEL_INFO);②

    zh = zookeeper_init(hostPort,③

                                main_watcher,
                                15000,
                                0,
                                0,
                                0);

    return errno;
}
```

①设置服务器ID。

②设置log日志的输出级别。

③创建会话的调用。

代码前两行设置了生成随机数的种子以及主节点的标识符，我们使用`server_id`来标识不同的主节点（回忆之前我们所介绍的，我们可以设置一或多个备份主节点和一个主要主节点）。之后我们设置了日志消息的输出级别，我们自己实现了日志功能（请见`log.h`），为了方便，我们将其拷贝到了ZooKeeper的发行包中（`zookeepe_log.h`）。最后，我们调用了`zookeeper_init`函数进行初始化，同时是`main_watcher`函数可以处理会话事件。

7.3 引导主节点

引导（**Bootstrapping**）主节点是指创建主从模式例子中使用的一些 **znode** 节点并竞选主要主节点的过程。我们首先创建四个必需的 **znode** 节点：

```
void bootstrap() {
    if(!connected) {①

        LOG_WARN(("Client not connected to ZooKeeper"));
        return;
    }
    create_parent("/workers", "");②

    create_parent("/assign", "");
    create_parent("/tasks", "");
    create_parent("/status", "");
    ...
}
```

①如果尚未连接，记录日志并退出。

②创建四个父节点： `/workers` 、 `/assign` 、 `/tasks` 、 `/status` 。

以下为 `create_parent` 函数：

```
void create_parent(const char * path,
                  const char * value) {
    zoo_create(zh, ①
```

path, ②

value, ③

0, ④

&ZOO_OPEN_ACL_UNSAFE, ⑤

0, ⑥

create_parent_completion, ⑦

NULL); ⑧

}

①异步调用创建znode节点时，需要传入的zhandle_t句柄的实例，在这个实现中，该实例为全局静态变量。

②该方法的path参数类型为const char*类型，path用于将客户端与对应的znode节点的子树连接在一起，详细信息参见10.3节。

③该函数的第三个参数为存储到znode节点的数据信息。我们通过create_parent函数传入了数据信息只是为了说明zoo_create函数需要传入数据信息的参数，而在我们的例子中，create_parent并不是必须传递数据信息的参数，因为本例中的四个节点的数据均为空值。

④该参数为保存数据信息（前一个参数）的长度值，本例中，设置为0。

⑤本例中，我们并不关心ACL策略问题，所以我们均设置为unsafe模式。

⑥这些znode节点为持久性的非有序节点，所以我们不需要传入任何标志位。

⑦该方法为异步调用，我们需要传入一个完成函数，ZooKeeper客户端会在操作请求完成时调用该函数。

⑧最后一个参数为上下文变量，本例中，不需要传入任何上下文变量。

因为该方法为异步调用，我们需要传入一个完成函数，ZooKeeper客户端会在操作请求完成时调用该函数，以下为完成函数的定义：

```
typedef void
    (*string_completion_t)(int rc, ①

                                const char *value, ②

                                const void *data); ③
```

①rc为返回码，在所有完成函数中都会返回该值。

②value返回值为字符串。

③data为在异步调用时传入的上下文变量数据，注意，开发人员负责该数据变量指针所指向的堆存储空间的内存释放。

在这个具体的例子中，我们的实现如下：

```
void create_parent_completion (int rc, const char *value, const void *data) {
    switch (rc) { ①

        case ZCONNECTIONLOSS:
            create_parent(value, (const char *) data); ②

        break;
        case ZOK:
            LOG_INFO(("Created parent node", value));
```

```
        break;
    case ZNODEEXISTS:
        LOG_WARN(("Node already exists"));
        break;
    default:
        LOG_ERROR(("Something went wrong when running for master"));
        break;
    }
}
```

①判断返回码来确定需要如何处理。

②在连接丢失时进行重试操作。

很多完成函数均包含简单的日志记录功能，以便告诉我们当前系统正在做什么。一般，完成函数会更加复杂，最好将各个完成函数的方法中的功能进行分离，就如我们在上面的例子中所展示的那样。注意，如果发生连接丢失的情况，上面的代码中，将会多次调用 `create_parent` 函数，该调用并非递归调用，因为完成函数并不是由 `create_parent` 所调用，而且，`create_parent` 函数只是简单的调用了 `ZooKeeper` 的函数，所以没有什么其他诸如内存空间分配等附加操作。如果在该函数中存某些附加操作，在每次调用前就需要先进行清理操作。

下一个任务为竞选主节点，竞选主节点主要就是尝试创建 `/master` 节点，以便锁定主要主节点角色。异步调用 `create` 方法与我们之前所讨论的有些不同，代码如下：

```
void run_for_master() {
    if(!connected) {
        LOG_WARN(LOGCALLBACK(zh),
```



```

        "Client not connected to ZooKeeper");
    return;
}
char server_id_string[9];
snprintf(server_id_string, 9, "%x", server_id);
zoo_acreate(zh,
            "/master",
            (const char *) server_id_string, ①

            sizeof(int), ②

            &ZOO_OPEN_ACL_UNSAFE,
            ZOO_EPHEMERAL, ③

            master_create_completion,
            NULL);
}

```

①在/master节点中保存服务器标识符信息。

②将数据的长度信息传入函数中，在这里，如我们所声明的，长度为一个int型的长度。

③该znode节点为临时性节点，因此我们必须传递临时性标志位参数。

到目前为止，完成函数也会比之前的版本更加复杂一些：

```

void master_create_completion (int rc, const char *value, const void *data) {
    switch (rc) {
        case ZCONNECTIONLOSS:
            check_master(); ①
    }
}

```

```

        break;
    case ZOK:
        take_leadership();②

        break;
    case ZNODEEXISTS:
        master_exists();③

        break;
    default:
        LOG_ERROR(LOGCALLBACK(zh),
                  "Something went wrong when running for master.");
        break;
    }
}

```

①连接丢失时，检查主节点的**znode**节点是否已经创建成功，或被其他主节点进程创建成功。

②如果我们成功创建了节点，就获得了管理权资格。

③如果主节点**znode**节点存在（其他进程已经创建并锁定该节点），就对该节点建立监视点，来监视该节点之后可能消失的情况。

如果该主节点进程发现**master**节点已经存在，就需要通过**zoo_awexists**方法来设置一个监视点：

```

void master_exists() {
    zoo_awexists(zh,
                 "/master",
                 master_exists_watcher,①

```

```
        NULL,  
        master_exists_completion, ②  
  
        NULL);  
}
```

①定义/**master**节点的监视点。

②该**exists**函数的回调方法的参数。

注意，我们可以在这个方法的调用中传入上下文对象，以便传给监视点对象使用，不过我们在这个例子中并未使用。我们可以在监视点函数中通过一个指向某个结构体或变量的（**void***）类型指针来传入上下文对象的变量。

在**znode**节点被删除时，我们会收到通知消息，以下代码为监视点函数处理通知消息的实现：

```
void master_exists_watcher (zhandle_t *zh,  
                           int type,  
                           int state,  
                           const char *path,  
                           void *watcherCtx) {if( type == ZOO_DELETED_EVENT) {  
assert( !strcmp(path, "/master") );      run_for_master(); ①  
  
    } else {      LOG_DEBUG(LOGCALLBACK(zh),      "Watched event:  
", type2string(type));    }}
```

①如果/**master**节点被删除，开始竞选主节点流程。

再回到我们之前的`master_exists`函数中，我们实现的完成函数很简单，也是我们目前为止一直采用的模式，细节上需要注意一点，即在执行创建/`master`节点操作和执行`exists`请求之间，也许/`master`节点已经被删除了（例如，前一个主要主节点进程已经退出），因此在完成函数中，我们还需要再次验证该`znode`节点是否存在，如果不存在，客户端进程需要再次竞选主节点流程：

```
void master_exists_completion (int rc,
                               const struct Stat *stat,
                               const void *data) {
    switch (rc) {
        case ZCONNECTIONLOSS:
        case ZOPERATIONTIMEOUT:
            master_exists();
            break;
        case ZOK:
            if(stat == NULL) {①

                LOG_INFO(LOGCALLBACK(zh),
                        "Previous master is gone, running for master");
                run_for_master();②

            }
            break;
        default:
            LOG_WARN(LOGCALLBACK(zh),
                    "Something went wrong when executing exists: ",
                    rc2string(rc));
            break;
    }
}
```

①通过判断返回的`stat`是否为`null`来检查该`znode`节点是否存在。

②如果节点不存在，再次进行主节点竞选的流程。

一旦主节点进程成为主要主节点，就可以开始行使管理权，我们将在下一节中进行说明。

7.4 行使管理权

一旦主节点进程被选定为主要主节点，它就可以开始行使其角色，首先它将获取所有可用从节点的列表信息：

```
void take_leadership() {
    get_workers();
}
void get_workers() {
    zoo_awget_children(zh,
                      "/workers",
                      workers_watcher, ①

                      NULL,
                      workers_completion, ②

                      NULL);
}
```

①设置一个监视点来监视从节点列表的变化情况。

②定义请求完成时需要调用的完成函数。

我们在实现中缓存了上一次读取的从节点列表信息，当我们再次读取到一个新的列表时，我们将会替换掉旧的列表，这些操作将在 `zoo_awget_children` 所指定的完成函数中完成：

```
void workers_completion (int rc,
                        const struct String_vector *strings,
```

```

                                const void *data) {
switch (rc) {
    case ZCONNECTIONLOSS:
    case ZOPERATIONTIMEOUT:
        get_workers();
        break;
    case ZOK:
        struct String_vector *tmp_workers =
            removed_and_set(strings, &workers);①

        free_vector(tmp_workers);②

        get_tasks();③

        break;
    default:
        LOG_ERROR(LOGCALLBACK(zh),
            "Something went wrong when checking workers: %s",
            rc2string(rc));
        break;
}
}

```

①更新从节点列表。

②我们的例子中，并没有真正使用这些从节点，所以在某个从节点被删除时，我们只是释放缓存资源。在重新分配任务的逻辑中，我们也采用该方式，只是用于练习的目的。

③下一步要进行的是任务的分配操作。

为了获取任务信息，服务端进程需要获取/tasks的所有子节点，并获取自上次读取后新引入的任务列表信息，所以我们每次读取后需要

区分获取的列表信息，否则可能会导致任务被分配两次（任务被分配两次也是有可能的，如果我们持有一个列表，而之后发生了两个连续的对/tasks节点的读操作并返回了一些重复的子节点元素，而此时，主节点在读取前没有足够的时间处理所有子节点元素，就可能导致无法区分某些子节点）。

```
void get_tasks () {
    zoo_awget_children(zh,
                        "/tasks",
                        tasks_watcher,
                        NULL,
                        tasks_completion,
                        NULL);
}
void tasks_watcher (zhandle_t *zh,
                    int type,
                    int state,
                    const char *path,
                    void *watcherCtx) {
    if( type == ZOO_CHILD_EVENT) {
        assert( !strcmp(path, "/tasks") );
        get_tasks();①
    } else {
        LOG_INFO(LOGCALLBACK(zh),
                 "Watched event: ",
                 type2string(type));
    }
}
void tasks_completion (int rc,
                       const struct String_vector *strings,
                       const void *data) {
    switch (rc) {
        case ZCONNECTIONLOSS:
        case ZOPERATIONTIMEOUT:
            get_tasks();
            break;
        case ZOK:
            LOG_DEBUG(LOGCALLBACK(zh), "Assigning tasks");
            struct String_vector *tmp_tasks = added_and_set(strings, &tasks);
            assign_tasks(tmp_tasks);②
    }
}
```



```
        free_vector(tmp_tasks);
        break;
    default:
        LOG_ERROR(LOGCALLBACK(zh),
                  "Something went wrong when checking tasks: %s",
                  rc2string(rc));
        break;
    }
}
```

①如果任务列表发生变化，再次获取任务列表信息。

②将还没有被分配的任务进行分配操作。

7.5 任务分配

任务分配包括任务信息的读取，选择一个从节点，通过在从节点任务列表中添加一个**znode**节点来分配该任务，最后将该任务从/**tasks**的子节点中删除。下面的代码实现了这些基本操作流程，获取任务信息、分配任务、删除任务这些操作均采用异步函数实现，提供所需的完成函数。代码如下所示：

```
void assign_tasks(const struct String_vector *strings) {
    int i;
    for( i = 0; i < strings->count; i++) {
        get_task_data( strings->data[i] );①

    }
}

void get_task_data(const char *task) {
    if(task == NULL) return;
    char * tmp_task = strdup(task, 15);
    char * path = make_path(2, "/tasks/", tmp_task);
    zoo_aget(zh,
            path,
            0,
            get_task_data_completion,
            (const void *) tmp_task);②

    free(path);
}

struct task_info {③

char* name;    char *value;    int value_len;    char* worker;};void
get_task_data_completion(int rc, const char *value, int value_len,
const struct Stat *stat, const void *data) {    int worker_index;    switch (rc)
{        case ZCONNECTIONLOSS:        case ZOPERATIONTIMEOUT:
```

```

get_task_data((const char *) data);          break;          case ZOK:
if(workers != NULL) {          worker_index = (rand() % workers->count);④

    struct task_info *new_task;⑤

    new_task = (struct task_info*) malloc(sizeof(struct task_info));
new_task->name = (char *) data;          new_task->value = strdup(value,
value_len);          new_task->value_len = value_len;          const
char * worker_string = workers->data[worker_index];          new_task-
>worker = strdup(worker_string);          task_assignment(new_task);⑥

    }          break;          default:
LOG_ERROR(LOGCALLBACK(zh),          "Something went wrong when
checking the master lock: %s",          rc2string(rc));
break;    }}

```

①对于每个任务，首先获取任务详细信息。

②异步调用来获取任务详细信息。

③用于保存任务上下文信息的结构体。

④随机选择一个从节点，将任务分配给这个从节点。

⑤创建一个新的**task_info**类型的变量来保存任务的详细信息。

⑥获取任务信息后，我们完成了任务分配操作。

到目前为止，以上代码已经完成了任务信息的读取和从节点的选择，下一步，需要创建用于表示任务分配的**znode**节点：

```

void task_assignment(struct task_info *task) {
    char* path = make_path(4, "/assign/" , task->worker, "/", task->name);
    zoo_acreate(zh,
                path,
                task->value,
                task->value_len,
                &ZOO_OPEN_ACL_UNSAFE,
                0,
                task_assignment_completion,
                (const void*) task);①

    free(path);
}
void task_assignment_completion (int rc, const char *value, const void *data) {
    switch (rc) {
        case ZCONNECTIONLOSS:
        case ZOPERATIONTIMEOUT:
            task_assignment((struct task_info*) data);
            break;
        case ZOK:
            if(data != NULL) {
                char * del_path = "";
                del_path = make_path(2, "/tasks/",
                                    ((struct task_info*) data)->name);
                if(del_path != NULL) {
                    delete_pending_task(del_path);②

                }
                free(del_path);
                free_task_info((struct task_info*) data);③
            }
            break;
        case ZNODEEXISTS:
            LOG_DEBUG(LOGCALLBACK(zh),
                    "Assignment has already been created: %s",
                    value);
            break;
        default:
            LOG_ERROR(LOGCALLBACK(zh),
                    "Something went wrong when checking the master lock: %s",
                    rc2string(rc));
            break;
    }
}
}

```

①创建表示任务分配的znode节点。

②一旦任务分配完成，主节点进程从待分配任务列表中删除这个任务。

③我们为task_info类型的实例分配了堆存储空间，因此我们现在可以释放该堆内存。

最后一步为删除/tasks下的任务节点，使/tasks节点下只有没有被分配的任务。

```
void delete_pending_task (const char * path) {
    if(path == NULL) return;
    char * tmp_path = strdup(path);
    zoo_adelete(zh,
                tmp_path,
                -1,
                delete_task_completion,
                (const void*) tmp_path);①

}

void delete_task_completion(int rc, const void *data) {
    switch (rc) {
        case ZCONNECTIONLOSS:
        case ZOPERATIONTIMEOUT:
            delete_pending_task((const char *) data);
            break;
        case ZOK:
            free((char *) data);②

            break;
        default:
            LOG_ERROR(LOGCALLBACK(zh),
                      "Something went wrong when deleting task: %s",
                      rc2string(rc));
            break;
    }
}
```

```
}    }
```

①异步删除任务节点。

②任务节点被成功删除后，没有什么其他需要做的，所以我们这里只是释放我们之前为存储路径字符串而分配的内存空间。

7.6 单线程与多线程客户端

ZooKeeper的发行包中，对于C语言组件提供了两个选项，多线程和单线程版本。我们建议开发者使用多线程版本，因为单线程版本只是因为一些历史原因。在Yahoo! 中，有些应用运行于BSD系统之中，而这些是单线程的应用，所以需要我们需要单线程版本的客户端库来使用ZooKeeper。如果你并未在这些强迫你采用单线程库的环境中，还是使用多线程版本。

使用单线程版本的库，你可以重用本章中介绍的这些代码，但需要额外实现一个事件循环操作，在我们的例子中，这个事件循环如下：

```
int initialized = 0;
int run = 0;
fd_set rfds, wfds, efds;
FD_ZERO(&rfds);
FD_ZERO(&wfds);
FD_ZERO(&efds);
while (!is_expired()) {
    int fd;
    int interest;
    int events;
    struct timeval tv;
    int rc;
    zookeeper_interest(zh, &fd, &interest, &tv);①

    if (fd != -1) {
        if (interest & ZOOKEEPER_READ) {②
```

```

        FD_SET(fd, &rfd);
    } else {
        FD_CLR(fd, &rfd);
    }
    if (interest & ZOOKEEPER_WRITE) {③

        FD_SET(fd, &wfd);
    } else {
        FD_CLR(fd, &wfd);
    }
} else {
    fd = 0;
}
/*
 * Master call to get a ZooKeeper handle.
 */
if(!initialized) {
    if(init(argv[1])) {④

        LOG_ERROR(("Error while initializing the master: ", errno));
    }
    initialized = 1;
}
/*
 * The next if block contains
 * calls to bootstrap the master
 * and run for master. We only
 * get into it when the client
 * has established a session and
 * is_connected is true.
 */
if(is_connected() && !run) {⑤

    LOG_INFO(("Connected, going to bootstrap and run for master"));
    /*
     * Create parent znodes
     */
    bootstrap();
    /*
     * Run for master
     */
    run_for_master();
    run = 1;
}
rc = select(fd+1, &rfd, &wfd, &efd, &tv);⑥

```



```

events = 0;
if (rc > 0) {
    if (FD_ISSET(fd, &rfdset)) {
        events |= ZOOKEEPER_READ;⑦
    }
    if (FD_ISSET(fd, &wfdset)) {
        events |= ZOOKEEPER_WRITE;⑧
    }
}
zookeeper_process(zh, events);⑨
}

```

①返回该客户端所关心的事件信息。

②添加ZOOKEEPER_READ事件到关注的事件集合中。

③添加ZOOKEEPER_WRITE事件到关注的事件集合中。

④此处为启动应用，与之前的7.2节介绍的init一样获取到一个ZooKeeper句柄。

⑤当主节点进程连接到ZooKeeper服务器后，就会收到ZOO_CONNECTED_EVENT事件，该部分会执行引导流程，并执行竞选主节点流程。

⑥采用select方式来等待新事件。

⑦指示文件描述符fd发生了读事件。

⑧指示文件描述符fd发生了写事件。

⑨处理其他ZooKeeper事件，zookeeper_process函数与之前的在多线程库中介绍的一样，需要处理监视事件和完成函数，在单线程版本中，我们必须自己处理这些事件。

这个事件循环将会关注相关的ZooKeeper事件，如回调和会话事件等。

使用多线程版本库时，编译客户端应用程序需要使用-l zookeeper_mt，并通过-DTHREADED定义THREADED宏选项。在代码中，在编译多线程库版本的程序时，通过THREADED宏定义会指示在调用执行时需要使用的代码片段。使用单线程版本库时，使用-l zookeeper_st来编译程序，同时不能指定-DTHREADED选项。在ZooKeeper的发行包中，你可以通过编译步骤指示来使用对应的库。

注意：阻塞回调函数

如果一个回调函数阻塞线程执行（比如，执行硬盘I/O操作），这样也许会导致会话超时，因为ZooKeeper处理循环时无法获取CPU事件

来执行会话操作，而在多线程版本库中不存在这个问题，因为 ZooKeeper 会使用单独的线程进行 I/O 的处理和完成函数的调用。

7.7 小结

ZooKeeper的C语言版本组件非常流行，在本章中我们介绍了如何使用C语言组件进行应用程序的开发。开发应用程序的流程与我们之前介绍的**Java**语言版本没有什么不同，关键的差异主要来自于语言之间的差异。例如，我们需要负责堆内存空间的管理，而在**Java**语言中我们将这些操作都委托给了**JVM**。同时我们还介绍了**ZooKeeper**所提供的多线程应用和单线程应用这两种实现方式，我们强烈建议开发者使用多线程版本，但是因为在发行包中也有单线程版本，我们也介绍了单线程版本开发库的使用。

第8章 Curator: ZooKeeper API的高级封装库

Curator作为ZooKeeper的一个高层次封装库，为开发人员封装了ZooKeeper的一组开发库，Curator的核心目标就是为你管理ZooKeeper的相关操作，将连接管理的复杂操作部分隐藏起来（理想上是隐藏全部）。我们在之前的内容中多次讨论了连接管理有多么棘手，通过Curator有时就可以顺利解决。

Curator为开发人员实现了一组常用的管理操作的菜谱，同时结合开发过程中的最佳实践和常见的边际情况的处理。例如，Curator实现了如锁（lock）、屏障（barrier）、缓存（cache）这些原语的菜谱，还实现了流畅（fluent）式的开发风格的接口。流畅式接口能够让我们将ZooKeeper中create、delete、getData等操作以流水线式的编程方式链式执行。同时，Curator还提供了命名空间（namespace）、自动重连和一些其他组件，使得应用程序更加健壮。

Curator组件最初由Netflix公司贡献并实现，而最近已经提升为Apache软件基金会的顶级项目。

本章将通过Curator来讲解如何实现我们之前的例子中的主节点程序。我们并不会大范围地详细讨论Curator，而是简单介绍Curator的功能，并将重点介绍Curator所提供的某些方便ZooKeeper应用程序使用

的特性。Curator的详细功能列表请访问该项目网站

(<http://curator.apache.org/>)。

8.1 Curator客户端程序

与使用ZooKeeper库开发时一样，使用Curator库进行开发，我们首先需要创建一个客户端实例，客户端实例为CuratorFramework类的实例对象，我们通过调用Curator提供的工厂方法来获得该实例：

```
CuratorFramework zkc =  
    CuratorFrameworkFactory.newClient(connectString, retryPolicy);
```

其中，connectString输入参数为我们将要连接的ZooKeeper服务器的列表，就像创建ZooKeeper客户端时一样。retryPolicy参数为Curator提供的新特性，通过这个参数，开发人员可以指定对于失去连接事件重试操作的处理策略。而在之前常规的ZooKeeper接口的开发中，在发生连接丢失事件时，我们往往需要再次提交操作请求。

注意：我们的例子中，实例化CuratorFramework类作为客户端。事实上，在工厂类中还提供了其他方法来创建实例，但我们并不详细讨论。其中有一个CuratorZooKeeperClient类，该类在ZooKeeper客户端实例上提供了某些附加功能，如保证请求操作在不可预见的连接断开情况下也能够安全执行，与CuratorFramework类不同，CuratorZooKeeperClient类中的操作执行与ZooKeeper客户端句柄直接相对应。

8.2 流畅式API

流畅式API可以让我们编写链式调用的代码，而不用在进行请求操作时采用严格的签名方案。例如，在标准的ZooKeeper的API中，我们同步创建一个znode节点的方法如下：

```
zk.create("/mypath",
          new byte[0],
          ZooDefs.Ids.OPEN_ACL_UNSAFE,
          CreateMode.PERSISTENT);
```

在Curator的流畅式API中，我们的调用方式如下：

```
zkc.create().withMode(CreateMode.PERSISTENT).forPath("/mypath", new byte[0]);
```

其中create调用返回一个CreateBuilder类的实例，随后调用的返回均为CreateBuilder类所继承的对象。例如，CreateBuilder继承了CreateModable<ACLBackgroundPathAndBytesable<String>>类，而withMode方法中声明了泛型接口CreateModable<T>。在Curator框架的客户端对象实例中，其他的如delete、getData、checkExists和getChildren方法也适用这种Builder模式。

对于异步的执行方法，我们只需要增加inBackground：

```
zkc.create().inBackground().withMode(CreateMode.PERSISTENT).forPath("/mypath",
    new byte[0]);
```

以上调用将会立刻返回，我们需要创建一或多个监听器来接收 **znode** 节点创建后的返回，在下一节中我们将介绍监听器，并介绍如何注册监听器。

很多方法可以实现异步调用的回调处理。假设在之前的调用中，回调方式将会通过 **CREATE** 事件传递给注册的监听器。**inBackground** 调用可以传入一个上下文对象，通过该参数可以传入一个具体的回调方法的实现，或是一个执行回调的执行器

（**java.util.concurrent.Executor**）。在Java中，执行器（**executor**）对象可以执行可运行的对象，我们可以通过执行器将回调方法的执行与 **ZooKeeper** 客户端线程的运行解耦，采用执行器常常比为每个任务新建一个线程更好。

为了设置监视点，我们只需要简单地在调用链中增加 **watched** 方法调用就可以，例如：

```
zkc.getData().inBackground().watched().forPath("/mypath");
```

上面设置的监视点将会通过监听器触发通知，这些通知将会以 **WATCHED** 事件传递给指定的监听器。我们还可以使用 **usingWathcer** 方法替换 **watched** 方法，**usingWathcer** 方法接受一个普通的 **ZooKeeper** 的 **Wathcer** 对象，并在接收到通知后调用该监视点方法。第三种选择就是

传入一个CuratorWatcher对象，CuratorWatcher的process方法与ZooKeeper的Watcher不同的是，它可能会抛出异常。


```
    }  
    } catch (Exception e) {  
        LOG.error("Exception while processing event.", e);  
        try {  
            close();  
        } catch (IOException ioe) {  
            LOG.error("IOException while closing.", ioe);  
        }  
    }  
};
```

因为我们只是为了说明监听器实现的结构，所以代码中每种事件的详细操作都被忽略了，对于代码细节，可以通过下载本书的代码示例来查看（<https://github.com/fpj/zookeeper-book-example>）。

之后，我们需要注册这个监听器，此时，我们需要一个框架客户端实例，创建方式可以采用我们之前所介绍的方式：

```
client = CuratorFrameworkFactory.newClient(hostPort, retryPolicy);
```

现在我们有框架客户端的实例，接下来注册监听器：

```
client.getCuratorListenable().addListener(masterListener);
```

还有一类比较特殊的监听器，这类监听器负责处理后台工作线程捕获的异常时的错误报告，该类监听器提供了底层细节的处理，不过，也许你在你的应用程序中需要处理这类问题。当应用程序需要处理这些问题时，就必须实现另一个监听器：

```
UnhandledErrorListener errorsListener = new UnhandledErrorListener() {  
    public void unhandledError(String message, Throwable e) {  
        LOG.error("Unrecoverable error: " + message, e);  
        try {
```

```
        close();
    } catch (IOException ioe) {
        LOG.warn( "Exception when closing.", ioe );
    }
};
```

同时，将该监听器注册到客户端实例中，如下所示：

```
client.getUnhandledErrorListenable().addListener(errorsListener);
```

注意，我们本节中所讨论的关于将监听器作为事件处理器的实现方式，与在之前的章节中所建议的**ZooKeeper**应用程序的实现不同（请见4.3节）。之前，我们采用链式调用和回调方法，而且每个回调方法都需要提供一个不同的回调实现，而在**Curator**实例中，回调方法或监视点通知这些细节均被封装为**Event**类，这也是更适合使用一个事件处理器的实现方式。

8.4 Curator中状态的转换

在Curator中暴露了与ZooKeeper不同的一组状态，比如SUSPENDED状态，还有Curator使用LOST来表示会话过期的状态。图8-1中展示了连接状态的状态机模型，当处理状态的转换时，我们建议将所有主节点操作请求暂停，因为我们并不知道ZooKeeper客户端能否在会话过期前重新连接，即使ZooKeeper客户端重新连接成功，也可能不再是主要主节点的角色，因此谨慎处理连接丢失的情况，对应用程序更加安全。

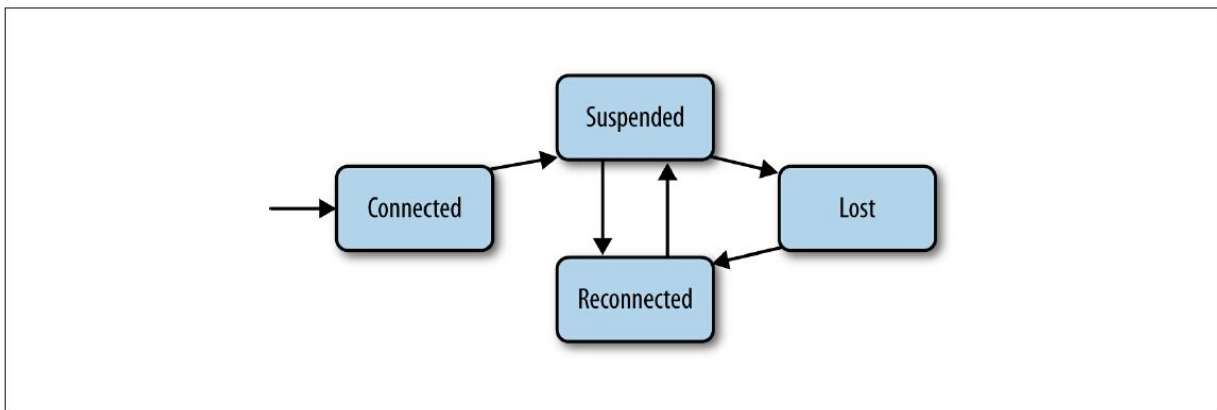


图8-1: Curator连接状态机模型

在我们的例子中，并未涉及的状态还有一个READ_ONLY状态，当ZooKeeper集群启用了只读模式，客户端所连接的服务器就会进入只读模式中，此时的连接状态也将进入只读模式。服务器转换到只读模式后，该服务器就会因隔离问题而无法与其他服务器共同形成仲裁的

最低法定数量，当连接状态为制度模式，客户端也将漏掉此时发生的任何更新操作，因为如果集群中存在一个子集的服务器数量，可以满足仲裁最低法定数量，并可以接收到客户端的对ZooKeeper的更新操作，还是会发生ZooKeeper的更新，也许这个子集的服务器会持续运行很久（ZooKeeper无法控制这种情况），那么漏掉的更新操作可能会无限多。漏掉更新操作的结果可能会导致应用程序的不正确的操作行为，所以，我们强烈建议启用该模式前仔细考虑其后果。注意，只读模式并不是Curator所独有的功能，而是通过ZooKeeper启用该选项（见第10章）。

8.5 两种边界情况

有两种有趣的错误场景，在Curator中都可以处理得很好，第一种是在有序节点的创建过程中发生的错误情况的处理，第二种为删除一个节点时的错误处理。

有序节点的情况

如果客户端所连接的服务器崩溃了，但还没来得及返回客户端所创建的有序节点的节点名称（即节点序列号），或者客户端只是连接丢失，客户端没接收到所请求操作的响应信息，结果，客户端并不知道所创建的znode节点路径名称。回忆我们对于有序节点的应用场景，例如，建立一个有序的所有客户端列表。为了解决这个问题，CreateBuilder提供了一个withProtection方法来通知Curator客户端，在创建的有序节点前添加一个唯一标识符，如果create操作失败了，客户端就会开始重试操作，而重试操作的一个步骤就是验证是否存在一个节点包含这个唯一标识符。

删除节点的保障

在进行delete操作时也可能发生类似情况，如果客户端在执行delete操作时，与服务器之间的连接丢失，客户端并不知道delete操作

是否成功执行。如果一个`znode`节点删除与否表示某些特殊情况，例如，表示一个资源处于锁定状态，因此确保该节点删除才能确保资源的锁定被释放，以便可以再次使用。`Curator`客户端中提供了一个方法，对应用程序的`delete`操作的执行提供了保障，`Curator`客户端会重新执行操作，直到成功为止，或`Curator`客户端实例不可用时。使用该功能，我们只需要使用`DeleteBuilder`接口中定义的`guaranteed`方法。

8.6 菜谱

Curator提供了很多种菜谱，建议你看一看这些可用的菜谱实现的列表。在我们的Curator主节点例子的实现中，用到了三种菜谱：

LeaderLatch、LeaderSelector和PathChildrenCache。

8.6.1 群首问

我们可以在应用程序中使用群首问（leader latch）这个原语进行主节点选举的操作。首先我们需要创建一个LeaderLatch的实例：

```
leaderLatch = new LeaderLatch(client, "/master", myId);
```

LeaderLatch的构造函数中，需要传入一个Curator框架客户端的实例，一个用于表示集群管理节点的群组的ZooKeeper路径，以及一个表示当前主节点的标识符。为了在Curator客户端获得或失去管理权时能够进行回调处理操作，我们需要注册一个LeaderLatchListener接口的实现，该接口中有两个方法：isLeader和notLeader。以下为isLeader实现的代码：

```
@Override
public void isLeader()
{
    ...
    /*
     * Start workersCache①
```

```

    */
workersCache.getListenable().addListener(workersCacheListener);
workersCache.start();
(new RecoveredAssignments(
    client.getZooKeeperClient().getZooKeeper()).recover(
    new RecoveryCallback() {
        public void recoveryComplete (int rc, List<String> tasks) {
            try {
                if(rc == RecoveryCallback.FAILED) {
                    LOG.warn("Recovery of assigned tasks failed.");
                } else {
                    LOG.info( "Assigning recovered tasks" );
                    recoveryLatch = new CountDownLatch(tasks.size());
                    assignTasks(tasks);②
                }
            }
        }
    })
    new Thread( new Runnable() {③

        public void run() {
            try {
                /*
                 * Wait until recovery is complete
                 */
                recoveryLatch.await();
                /*
                 * Start tasks cache
                 */
                tasksCache.getListenable().
                    addListener(tasksCacheListener);④

                tasksCache.start();
            } catch (Exception e) {
                LOG.warn("Exception while assigning
                    and getting tasks.",
                        e );
            }
        }
    }).start();
} catch (Exception e) {
    LOG.error("Exception while executing the recovery callback",
        e);
}
}

```

```
    });  
}
```

①我们首先初始化一个从节点缓存列表的实例，以确保有可以分配任务的从节点。

②一旦发现存在之前的主节点没有分配完的任务需要分配，我们将继续进行任务分配。

③我们实现了一个任务分配的屏障，这样我们就可以在开始分配新任务前，等待已恢复的任务的分配完成，如果我们不这样做，新的主节点会再次分配所有已恢复的任务。我们启动了一个单独的线程进行处理，以便不会锁住ZooKeeper客户端回调线程的运行。

④当主节点完成恢复任务的分配操作，我们开始进行新任务的分配操作。

我们实现的这个方法作为CuratorMasterLatch类的一部分，而CuratorMasterLatch类为LeaderLatchListener接口的实现类，我们需要在具体流程开始前注册监听器。我们在runForMaster方法中进行这两步操作，同时，我们还将注册另外两个监听器，来处理事件的监听和错误：

```
public void runForMaster() {  
    client.getCuratorListenable().addListener(masterListener);  
    client.getUnhandledErrorListenable().addListener(errorsListener);  
    leaderLatch.addListener(this);  
    leaderLatch.start();  
}
```

对于notLeader方法，我们会在主节点失去管理权时进行调用，在本例中，我们只是简单地关闭了所有对象实例，对这个例子来说，这些操作已经足够了。在实际的应用程序中，你也许还需要进行某些状态的清理操作并等待再次成为主节点。如果LeaderLatch对象没有关闭，Curator客户端有可能再次获得管理权。

8.6.2 群首选举器

选举主节点时还可以使用的另一个菜谱为LeaderSelector。LeaderSelector和LeaderLatch之间主要区别在于使用的监听器接口不同，其中LeaderSelector使用了LeaderSelectorListener接口，该接口中定义了takeLeadership方法，并继承了stateChanged方法，我们可以在我们的应用程序中使用群首选举原语来进行一个主节点的选举操作，首先我们需要创建一个LeaderSelector实例。

```
leaderSelector = new LeaderSelector(client, "/master", this);
```

LeaderSelector的构造函数中，接受一个Curator框架客户端实例，一个表示该主节点所参与的集群管理节点群组的ZooKeeper路径，以及一个LeaderSelectorListener接口的实现类的实例。集群管理节点群组表示所有参与主节点选举的Curator客户端。在LeaderSelectorListener的实现中必须包含takeLeadership方法和stateChanged方法，其中

takeLeadership方法用于获取管理权，在我们的例子中，该代码实现与isLeader类似，以下为我们实现的takeLeadership方法：

```
CountDownLatch leaderLatch = new CountDownLatch(1);
CountDownLatch closeLatch = new CountDownLatch(1);

@Override
public void takeLeadership(CuratorFramework client) throws Exception
{
    ...
    /*
     * Start workersCache
     */
    workersCache.getListenable().addListener(workersCacheListener);
    workersCache.start();
    (new RecoveredAssignments(
        client.getZooKeeperClient().getZooKeeper()).recover(
        new RecoveryCallback() {
            public void recoveryComplete (int rc, List<String> tasks) {
                try {
                    if(rc == RecoveryCallback.FAILED) {
                        LOG.warn("Recovery of assigned tasks failed.");
                    } else {
                        LOG.info( "Assigning recovered tasks" );
                        recoveryLatch = new CountDownLatch(tasks.size());
                        assignTasks(tasks);
                    }
                }
                new Thread( new Runnable() {
                    public void run() {
                        try {
                            /*
                             * Wait until recovery is complete
                             */
                            recoveryLatch.await();
                            /*
                             * Start tasks cache
                             */
                            tasksCache.getListenable().
                                addListener(tasksCacheListener);
                            tasksCache.start();
                        } catch (Exception e) {
                            LOG.warn("Exception while assigning
                                and getting tasks.",
                                    e );
                        }
                    }
                }).start();
            }
        }).start();
    /*
     * Decrement latch
     */
    leaderLatch.countDown();①
```

```

        } catch (Exception e) {
            LOG.error("Exception while executing the recovery callback",
                e);
        }
    }
});
/*
 * This latch is to prevent this call from exiting. If we exit, then
 * we release mastership.
 */
closeLatch.await();②
}

```

①我们通过一个单独的CountDownLatch原语来等待该Curator客户端获取管理权。

②如果主节点退出了takeLeadership方法，也就放弃了管理权，我们通过CountDownLatch来阻止退出该方法，直到主节点关闭为止。

我们实现的这个方法为CuratorMaster类的一部分，而CuratorMaster类实现了LeaderSelectorListener接口。对于主节点来说，如果想要释放管理权只能退出takeLeadership方法，所以我们需要通过某些锁等机制来阻止该方法的退出，在我们的实现中，我们在退出主节点时通过递减门（latch）值来实现。

我们依然在runForMaster方法中启动我们的主节点选择器，与LeaderLatch的方式不同，我们不需要注册一个监听器（因为我们在构

构造函数中已经注册了监听器）：

```
public void runForMaster() {
    client.getCuratorListenable().addListener(masterListener);
    client.getUnhandledErrorListenable().addListener(errorsListener);
    leaderSelector.setId(myId);
    leaderSelector.start();
}
```

另外我们还需要给这个主节点一个任意的标识符，虽然我们在本例中并未实现，但我们可以设置群首选择器在失去管理权后自动重新排队（`LeaderSelector.autoRequeue`）。重新排队意味着该客户端会一直尝试获取管理权，并在获得管理权后执行`takeLeadership`方法。

作为`LeaderSelectorListener`接口实现的一部分，我们还实现了一个处理连接状态变化的方法：

```
@Override
public void stateChanged(CuratorFramework client, ConnectionState newState)
{
    switch(newState) {
        case CONNECTED:
            //Nothing to do in this case.
            break;
        case RECONNECTED:
            // Reconnected, so I should①

            // still be the leader.
            break;
        case SUSPENDED:
            LOG.warn("Session suspended");
            break;
        case LOST:
            try {
                close();②
            }
    }
}
```



```
        } catch (IOException e) {
            LOG.warn( "Exception while closing", e );
        }
        break;
    case READ_ONLY:
        // We ignore this case.
        break;
    }
}
```

①所有操作均需要通过ZooKeeper集群实现，因此，如果连接丢失，主节点也就无法先进行任何操作请求，因此在这里我们最好什么都不做。

②如果会话丢失，我们只是关闭这个主节点程序。

8.6.3 子节点缓存器

我们在示例中使用的最后一个菜谱是子节点缓存器（PathChildrenCached类）。我们将使用该类保存从节点的列表和任务列表，该缓存器负责保存一份子节点列表的本地拷贝，并会在该列表发生变化时通知我们。注意，因为时间问题，也许在某些特定时间点该缓存的数据集合与ZooKeeper中保存的信息并不一致，但这些变化最终都会反映到ZooKeeper中。

为了处理每一个缓存器实例的变化情况，我们需要一个PathChildrenCacheListener接口的实现类，该接口中只有一个方法childEvent。对于从节点信息的列表，我们只关心从节点离开的情况，

因为我们需要重新分配已经分给这些节点的任务，而列表中添加信息对于分配新任务更加重要：

```
PathChildrenCacheListener workersCacheListener = new PathChildrenCacheListener()
{
    public void childEvent(CuratorFramework client, PathChildrenCacheEvent event)
    {
        if(event.getType() == PathChildrenCacheEvent.Type.CHILD_REMOVED) {
            /*
             * Obtain just the worker's name
             */
            try {
                getAbsentWorkerTasks(event.getData().getPath().replaceFirst(
                    "/workers/", ""));
            } catch (Exception e) {
                LOG.error("Exception while trying to re-assign tasks.", e);
            }
        }
    }
};
```

对于任务列表，我们通过列表增加的情况来触发任务分配的过程：

```
PathChildrenCacheListener tasksCacheListener = new PathChildrenCacheListener() {
    public void childEvent(CuratorFramework client, PathChildrenCacheEvent
        event) {
        if(event.getType() == PathChildrenCacheEvent.Type.CHILD_ADDED) {
            try {
                assignTask(event.getData().getPath().replaceFirst("/tasks/", ""));
            } catch (Exception e) {
                LOG.error("Exception when assigning task.", e);
            }
        }
    }
};
```

注意，我们这里假设至少有一个可用的从节点可以分配任务给它，当前没有可用的从节点时，我们需要暂停任务分配，并保存列表信息的增加信息，以便在从节点列表中新增可用从节点时可以将这些

没有分配的任务进行分配。为了简单起见，我们并没有实现这一功能，读者可以作为练习自己实现。

8.7 小结

Curator实现了一系列很不错的ZooKeeper API的扩展，将ZooKeeper的复杂性进行了抽象，并实现了在实际生产环境中经验的最佳实践和社区讨论的某些特性。在本章中，我们通过我们的主从模式的例子，描述了如何通过Curator所提供的功能来实现一个主节点角色的程序，我们用到了群首选举的实现和子节点缓存器，通过这些我们实现了主节点中的重要特性。这两个菜谱并不是Curator所提供的所有特性，还有很多其他菜谱和特性可以供我们使用。

第三部分 ZooKeeper的管理

本书该部分将会介绍有关ZooKeeper的管理的相关信息。通过深入ZooKeeper的内部运行原理提供给你相关的管理背景，使你可以在关键问题上做出正确的选择，例如你需要多少台ZooKeeper服务器，你如何调整这些服务器之间的通信。

第9章 ZooKeeper内部原理

本章与其他章节不同，本章不会讲解任何关于如何通过ZooKeeper构建一个应用程序相关的知识，主要是介绍ZooKeeper内部是如何运行的，通过从高层次介绍其所使用的协议，以及ZooKeeper所采用的在提供高性能的同时还具有容错能力的机制。这些内容非常重要，通过这些为大家提供了一个更深度的视角来分析为什么程序与ZooKeeper一同工作时是如此运行。如果你打算运行ZooKeeper，该视角对你会非常有用，同时本章也是下一章的背景知识。

我们在前几章中已经了解，ZooKeeper运行于一个集群环境中，客户端会连接到这些服务器执行操作请求，但究竟这些服务器对客户端所发送的请求操作做了哪些工作？我们在第2章中已经提到了，我们选择某一个服务器，称之为群首（leader）。其他服务器追随群首，被称为追随者（follower）。群首作为中心点处理所有对ZooKeeper系统变更的请求，它就像一个定序器，建立了所有对ZooKeeper状态的更新的顺序，追随者接收群首所发出更新操作请求，并对这些请求进行处理，以此来保障状态更新操作不会发生碰撞。

群首和追随者组成了保障状态变化有序的核心实体，同时还存在第三类服务器，称为观察者（observer）。观察者不会参与决策哪些请

求可被接受的过程，只是观察决策的结果，观察者的设计只是为了系统的可扩展性。

本章中，我们还会介绍我们用于实现ZooKeeper集群和服务器与客户端内部通信所使用的协议。我们首先以客户端的请求和事务的这些常见概念展开讨论，这些概念将贯穿本章后续部分。

注意：代码参考

因为本章涉及系统内部运行原理，我们认为提供代码的参考会更有用，因此通过源代码与本章中的内容相对应来阅读本章，更好的对应其中的类和方法。

9.1 请求、事务和标识符

ZooKeeper服务器会在本地处理只读请求（`exists`、`getData`和`getChildren`）。假如一个服务器接收到客户端的`getData`请求，服务器读取该状态信息，并将这些信息返回给客户端。因为服务器会在本地处理请求，所以ZooKeeper在处理以只读请求为主要负载时，性能会很高。我们还可以增加更多的服务器到ZooKeeper集群中，这样就可以处理更多的读请求，大幅提高整体处理能力。

那些会改变ZooKeeper状态的客户端请求（`create`、`delete`和`setData`）将会被转发给群首，群首执行相应的请求，并形成状态的更新，我们称为事务（`transaction`）。其中，请求表示源自于客户端发起的操作，而事务则包含了对应请求处理而改变ZooKeeper状态所需要执行的步骤。我们通过一个简单点的例子，而不是ZooKeeper的操作，来说明这个问题。假如，操作为`inc (i)`，该方法对变量`i`的值进行增量操作，如果此时一个请求为`inc (i)`，假如`i`的值为10，该操作执行后，其值为11。再回过头看请求和事务的概念，其中`inc (i)`为请求，而事务则为变量`i`和11（变量`i`保存了11这个值）。

现在我们再来看看ZooKeeper的例子，假如一个客户端提交了一个对`/z`节点的`setData`请求，`setData`将会改变该`znode`节点数据信息，并会

增加该节点版本号，因此，对于这个请求的事务包括了两个重要字段：节点中新的数据字段值和该节点新的版本号。当处理该事务时，服务端将会用事务中的数据信息来替换/z节点中原来的数据信息，并用事务中的版本号更新该节点，而不是增加版本号的值。

一个事务为一个单位，也就是说所有的变更处理需要以原子方式执行。以setData的操作为例，变更节点的数据信息，但并不改变版本号将会导致错误的发生，因此，ZooKeeper集群以事务方式运行，并确保所有的变更操作以原子方式被执行，同时不会被其他事务所干扰。在ZooKeeper中，并不存在传统的关系数据库中所涉及的回滚机制，而是确保事务的每一步操作都互不干扰。在很长的一段时间里，ZooKeeper所采用的设计方式为，在每个服务器中启动一个单独的线程来处理事务，通过单独的线程来保障事务之间的顺序执行互不干扰。最近，ZooKeeper增加了多线程的支持，以便提高事务处理的速度。

同时一个事务还具有幂等性，也就是说，我们可以对同一个事务执行两次，我们得到的结果还是一样的，我们甚至还可以对多个事务执行多次，同样也会得到一样的结果，前提是我们确保多个事务的执行顺序每次都是一样的。事务的幂等性可以让我们在进行恢复处理时更加简单。

当群首产生了一个事务，就会为该事务分配一个标识符，我们称之为ZooKeeper会话ID（zxid），通过Zxid对事务进行标识，就可以按

照群首所指定的顺序在各个服务器中按序执行。服务器之间在进行新的群首选举时也会交换zxid信息，这样就可以知道哪个无故障服务器接收了更多的事务，并可以同步他们之间的状态信息。

zxid为一个long型（64位）整数，分为两部分：时间戳（epoch）部分和计数器（counter）部分。每个部分为32位，在我们讨论zab协议时，我们就会发现时间戳（epoch）和计数器（counter）的具体作用，我们通过该协议来广播各个服务器的状态变更信息。

9.2 群首选举

群首为集群中的服务器选择出来的一个服务器，并会一直被集群所认可。设置群首的目的是为了对客户端所发起的ZooKeeper状态变更请求进行排序，包括：`create`、`setData`和`delete`操作。群首将每一个请求转换为一个事务，如前一节中所介绍，将这些事务发送给追随者，确保集群按照群首确定的顺序接受并处理这些事务。

为了了解管理权的原理，一个服务器必须被仲裁的法定数量的服务器所认可。在第2章中我们已经讨论过，法定数量必须集群数量是能够交错在一起，以避免我们所说的脑裂问题（`split brain`）：即两个集合的服务器分别独立的运行，形成了两个集群。这种情况将导致整个系统状态的不一致性，最终客户端也将根据其连接的服务器而获得不同的结果，在2.3.3节我们已经通过具体例子说明了这一情况。

选举并支持一个群首的集群服务器数量必须至少存在一个服务器进程的交叉，我们使用属于仲裁（`quorum`）来表示这样一个进程的子集，仲裁模式要求服务器之间两两相交。

注意：进展

一组服务器达到仲裁法定数量是必需条件，如果足够多的服务器永久性地退出，无法达到仲裁法定数量，ZooKeeper也就无法取得进

展。即使服务器退出后再次启动也可以，但必须保证仲裁的法定数量的服务器最终运行起来。我们先不讨论这个问题，而在下一章中再讨论重新配置集群，重新配置可以随时间而改变仲裁法定数量。

每个服务器启动后进入LOOKING状态，开始选举一个新的群首或查找已经存在的群首，如果群首已经存在，其他服务器就会通知这个新启动的服务器，告知哪个服务器是群首，与此同时，新的服务器会与群首建立连接，以确保自己的状态与群首一致。

如果集群中所有的服务器均处于LOOKING状态，这些服务器之间就会进行通信来选举一个群首，通过信息交换对群首选举达成共识的选择。在本次选举过程中胜出的服务器将进入LEADING状态，而集群中其他服务器将会进入FOLLOWING状态。

对于群首选举的消息，我们称之为群首选举通知消息（leader election notifications），或简单地称为通知（notifications）。该协议非常简单，当一个服务器进入LOOKING状态，就会发送向集群中每个服务器发送一个通知消息，该消息中包括该服务器的投票（vote）信息，投票中包含服务器标识符（sid）和最近执行的事务的zxid信息，比如，一个服务器所发送的投票信息为（1，5），表示该服务器的sid为1，最近执行的事务的zxid为5（出于群首选举的目的，zxid只有一个数字，而在其他协议中，zxid则有时间戳epoch和计数器组成）。

当一个服务器收到一个投票信息，该服务器将会根据以下规则修改自己的投票信息：

- 1.将接收的`voteId`和`voteZxid`作为一个标识符，并获取接收方当前的投票中的`zxid`，用`myZxid`和`mySid`表示接收方服务器自己的值。

- 2.如果（`voteZxid > myZxid`）或者（`voteZxid = myZxid`且`voteId > mySid`），保留当前的投票信息。

- 3.否则，修改自己的投票信息，将`voteZxid`赋值给`myZxid`，将`voteId`赋值给`mySid`。

简而言之，只有最新的服务器将赢得选举，因为其拥有最近一次的`zxid`。我们稍后会看到，这样做将会简化群首崩溃后重新仲裁的流程。如果多个服务器拥有最新的`zxid`值，其中的`sid`值最大的将赢得选举。

当一个服务器接收到仲裁数量的服务器发来的投票都一样时，就表示群首选举成功，如果被选举的群首为某个服务器自己，该服务器将会开始行使群首角色，否则就成为一个追随者并尝试连接被选举的群首服务器。注意，我们并未保证追随者必然会成功连接上被选举的群首服务器，比如，被选举的群首也许此时崩溃了。一旦连接成功，追随者和群首之间将会进行状态同步，在同步完成后，追随者才可以处理新的请求。

注意：查找群首

在ZooKeeper中对应的实现选举的Java类为QuorumPeer，其中的run方法实现了服务器的主要工作循环。当进入LOOKING状态，将会执行lookForLeader方法来进行群首的选举，该方法主要执行我们刚刚所讨论的协议，该方法返回前，在该方法中会将服务器状态设置为LEADING状态或FOLLOWING状态，当然还可能为OBSERVING状态，我们稍后讨论这个状态。如果服务器成为群首，就会创建一个Leader对象并运行这个对象，如果服务器为追随者，就会创建一个Follower对象并运行。

现在，让我们通过例子来重温这个协议的执行过程。图9-1展示了三个服务器，这三个服务器分别以不同的初始投票值开始，其投票值取决于该服务器的标识符和其最新的zxid。每个服务器会收到另外两个服务器发送的投票信息，在第一轮之后，服务器s2和服务器s3将会改变其投票值为（1，6），之后服务器s2和服务器s3在改变投票值之后会发送新的通知消息，在接收到这些新的通知消息后，每个服务器收到的仲裁数量的通知消息拥有一样的投票值，最后选举出服务器s1为群首。

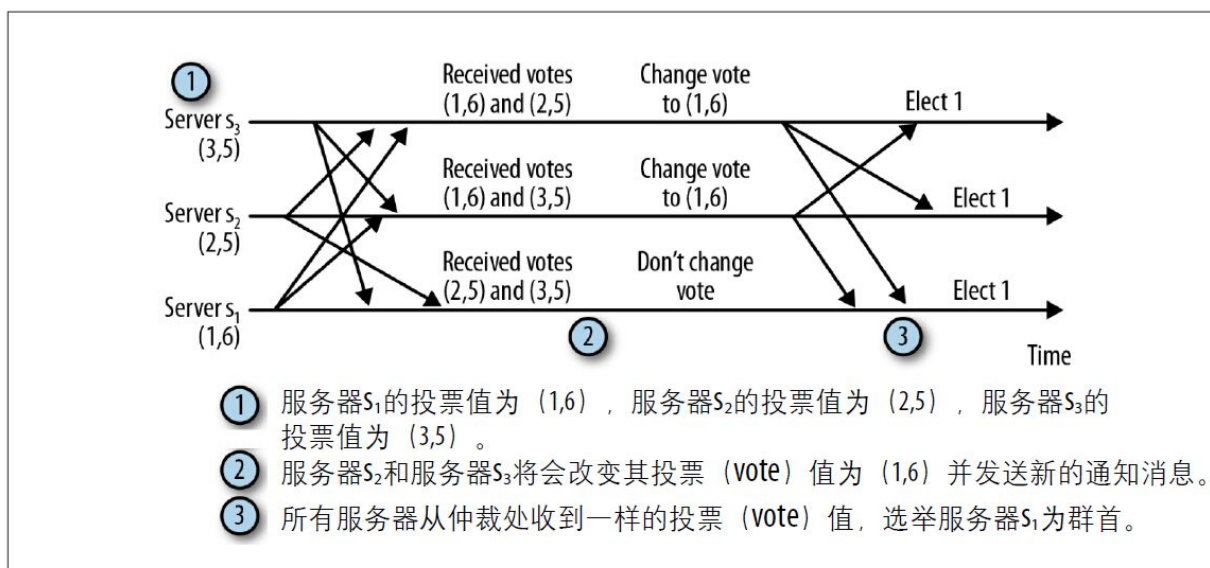


图9-1: 群首选举过程的示例

并不是所有执行过程都如图9-1中所示, 在图9-2中, 我们展示了另一种情况的例子。服务器 s_2 做出了错误判断, 选举了另一个服务器 s_3 而不是服务器 s_1 , 虽然 s_1 的zxid值更高, 但在从服务器 s_1 向服务器 s_2 传递消息时发生了网络故障导致长时间延迟, 与此同时, 服务器 s_2 选择了服务器 s_3 作为群首, 最终, 服务器 s_1 和服务器 s_3 组成了仲裁数量 (quorum), 并将忽略服务器 s_2 。

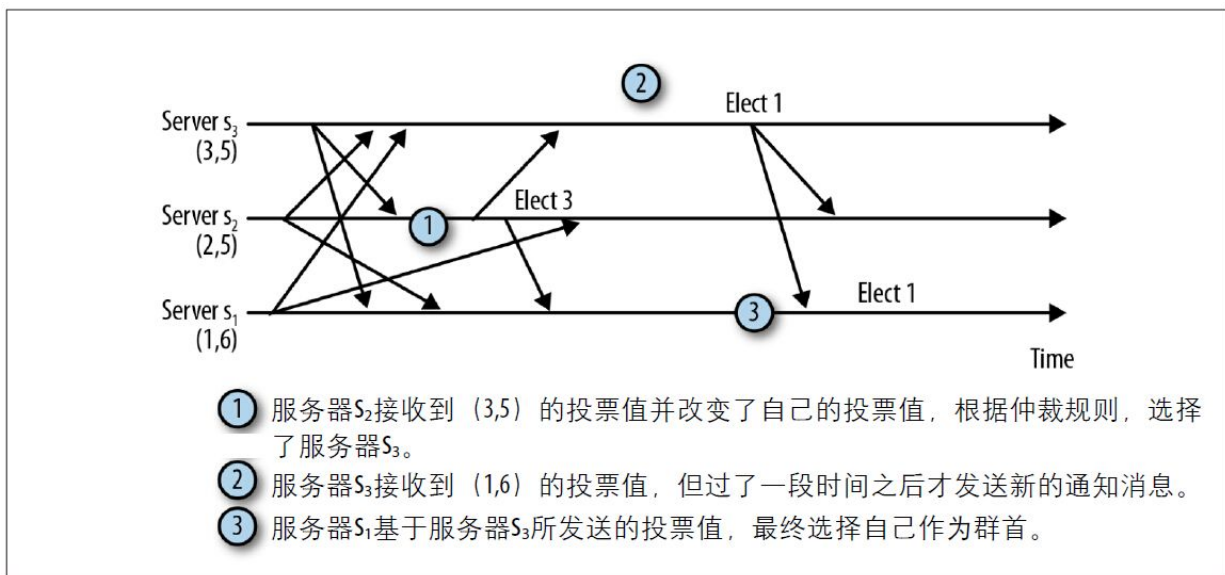


图9-2：消息交错导致一个服务器选择了另一个群首

虽然服务器 s_2 选择了另一个群首，但并未导致整个服务发生错误，因为服务器 s_3 并不会以群首角色响应服务器 s_2 的请求，最终服务器 s_2 将会在等待被选择的群首 s_3 的响应时而超时，并开始再次重试。再次尝试，意味着在这段时间内，服务器 s_2 无法处理任何客户端的请求，这样做并不可取。

从这个例子，我们发现，如果让服务器 s_2 在进行群首选举时多等待一会，它就能做出正确的判断。我们通过图9-3展示这种情况，我们很难确定服务器需要等待多长时间，在现在的实现中，默认的群首选举的实现类为FastLeaderElection，其中使用固定值200ms（常量finalizeWait），这个值比在当今数据中心所预计的长消息延迟（不到1毫秒到几毫秒的时间）要长得多，但与恢复时间相比还不够长。万一此类延迟（或任何其他延迟）时间并不是很长，一个或多个服务器最

终将错误选举一个群首，从而导致该群首没有足够的追随者，那么服务器将不得不再次进行群首选举。错误地选举一个群首可能会导致整个恢复时间更长，因为服务器将会进行连接以及不必要的同步操作，并需要发送更多消息来进行另一轮的群首选举。

注意：快速群首选举的快速指的是什么？

如果你想知道为什么我们称当前默认的群首选举算法为快速算法，这个问题有历史原因。最初的群首选举算法的实现采用基于拉取式的模型，一个服务器拉取投票值的间隔大概为1秒，该方法增加了恢复的延迟时间，相比较现在的实现方式，我们可以更加快速地进行群首选举。

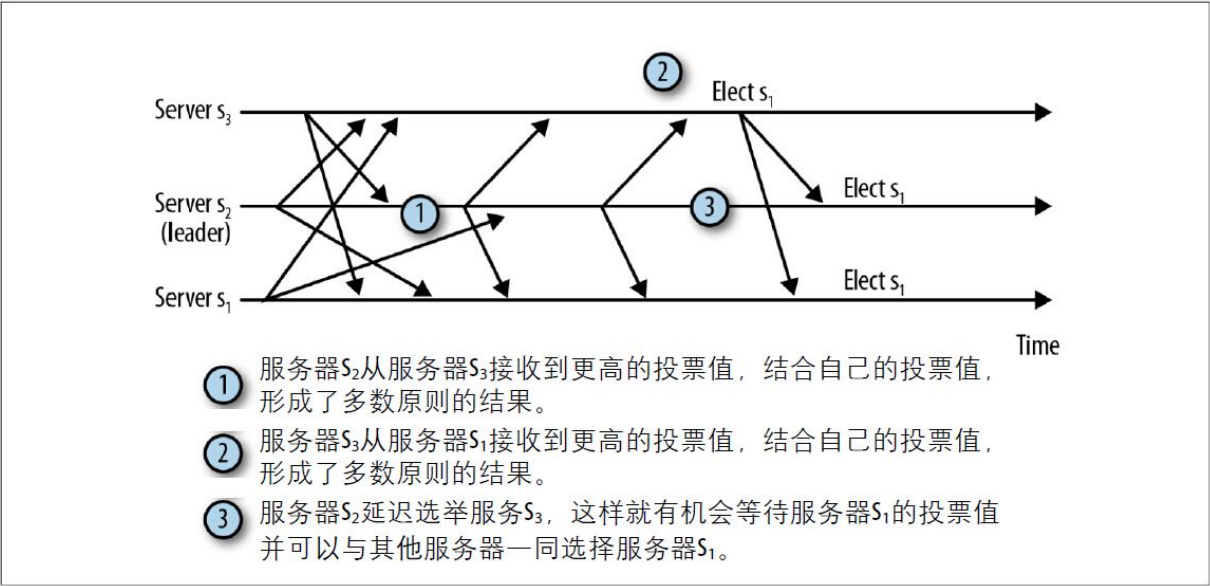


图9-3：群首选举时的长延迟

如果想实现一个新的群首选举的算法，我们需要实现一个quorum包中的Election接口。为了可以让用户自己选择群首选举的实现，代码中使用了简单的整数标识符（请查看代码中QuorumPeer.createElectionAlgorithm（）），另外两种可选的实现方式为LeaderElection类和AuthFastLeaderElection类，但在版本3.4.0中，这些类已经标记为弃用状态，因此，在未来的发布版本中，你可能不会再看到这些类。

9.3 Zab: 状态更新的广播协议

在接收到一个写请求操作后，追随者会将请求转发给群首，群首将探索性地执行该请求，并将执行结果以事务的方式对状态更新进行广播。一个事务中包含服务器需要执行变更的确切操作，当事务提交时，服务器就会将这些变更反馈到数据树上，其中数据树为ZooKeeper用于保存状态信息的数据结构（请参考DataTree类）。

之后我们需要面对的问题便是服务器如何确认一个事务是否已经提交，由此引入了我们所采用的协议：**Zab: ZooKeeper原子广播协议**（**ZooKeeper Atomic Broadcast protocol**）。假设现在我们有一个活动的群首服务器，并拥有仲裁数量的追随者支持该群首的管理权，通过该协议提交一个事务非常简单，类似于一个两阶段提交。

- 1.群首向所有追随者发送一个**PROPOSAL**消息 p 。
- 2.当一个追随者接收到消息 p 后，会响应群首一个**ACK**消息，通知群首其已接受该提案（**proposal**）。
- 3.当收到仲裁数量的服务器发送的确认消息后（该仲裁数包括群首自己），群首就会发送消息通知追随者进行提交（**COMMIT**）操作。

图9-4说明了这一个过程的具体步骤顺序，我们假设群首通过隐式方式给自己发送消息。

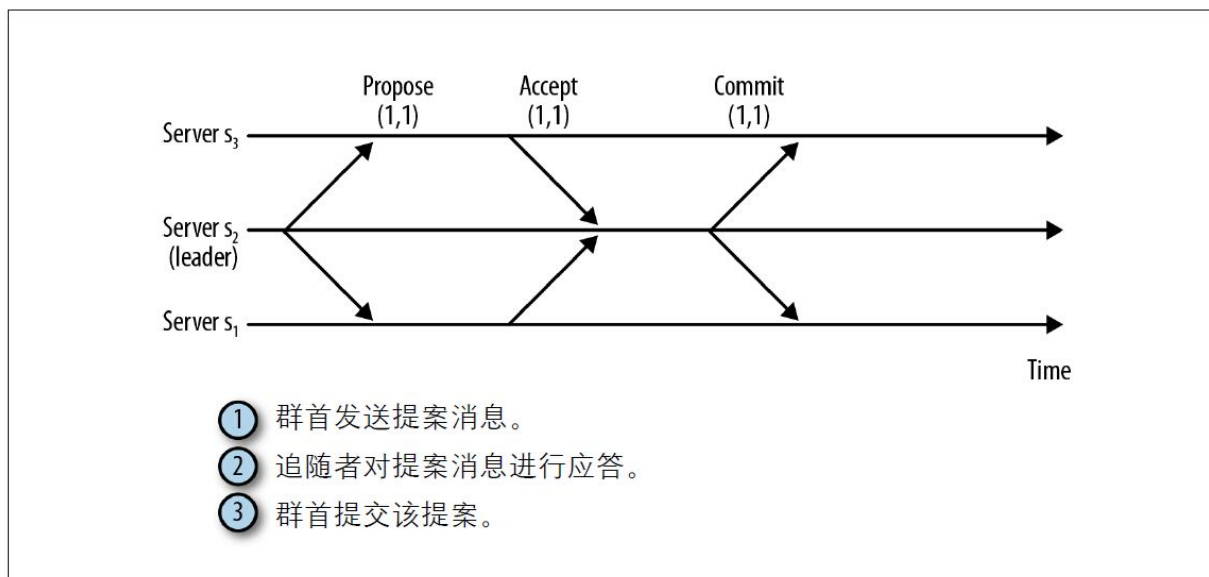


图9-4：提交提案的常规消息模式

在应答提案消息之前，追随者还需要执行一些检查操作。追随者将会检查所发送的提案消息是否属于其所追随的群首，并确认群首所广播的提案消息和提交事务消失的顺序正确。

Zab保障了以下几个重要属性：

·如果群首按顺序广播了事务T和事务T，那么每个服务器在提交T？事务前保证事务T已经提交完成。

·如果某个服务器按照事务T、事务T的顺序提交事务，所有其他服务器也必然会在提交事务T前提交事务T。

第一个属性保证事务在服务器之间的传送顺序的一致，而第二个竖向地保证服务器不会跳过任何事务。假设事务为状态变更操作，每个状态变更操作又依赖前一个状态变更操作的结果，如果跳过事务就会导致结果的不一致性，而两阶段提交保证了事务的顺序。**Zab**在仲裁数量服务器中记录了事务，集群中仲裁数量的服务器需要在群首提交事务前对事务达成一致，而且追随者也会在硬盘中记录事务的确认信息。

我们在9.6节将会看到，事务在某些服务器上可能会终结，而其他服务器上却不会，因为在写入事务到存储中时，服务器也可能发生崩溃。无论何时，只要仲裁条件达成并选举了一个新的群首，**ZooKeeper**都可以将所有服务器的状态更新到最新。

但是，**ZooKeeper**自始至终并不总是有一个活动的群首，因为群首服务器也可能崩溃，或短时间地失去连接，此时，其他服务器需要选举一个新的群首以保证系统整体仍然可用。其中时间戳（**epoch**）的概念代表了管理权随时间的变化情况，一个时间戳表示了某个服务器行使管理权的这段时间，在一个时间戳内，群首会广播提案消息，并根据计数器（**counter**）识别每一个消息。我们知道**zxid**的第一个元素为时间戳信息，因此每个**zxid**可以很容易地与事务被创建时间戳相关联。

时间戳的值在每次新群首选举发生的时候便会增加。同一个服务器成为群首后可能持有不同的时间戳信息，但从协议的角度出发，一

个服务器行使管理权时，如果持有不同的时间戳，该服务器就会被认为是不同的群首。如果服务器s成为群首并且持有的时间戳为4，而当前已经建立的群首的时间戳为6，集群中的追随者会追随时间戳为6的群首s，处理群首在时间戳6之后的消息。当然，追随者在恢复阶段也会接收时间戳4到时间戳6之间的提案消息，之后才会开始处理时间戳为6之后的消息，而实际上这些提案消息是以时间戳6的消息来发送的。

在仲裁模式下，记录已接收的提案消息非常关键，这样可以确保所有的服务器最终提交了被某个或多个服务已经提交完成的事务，即使群首在此时发生了故障。完美检测群首（或任何服务器）是否发生故障是非常困难的，虽然不是不可能，但在很多设置的情况下，都可能发生对一个群首是否发生故障的错误判断。

实现这个广播协议所遇到最多的困难在于群首并发存在情况的出现，这种情况并不一定是脑裂场景。多个并发的群首可能会导致服务器提交事务的顺序发生错误，或者直接跳过了某些事务。为了阻止系统中同时出现两个服务器自认为自己是群首的情况是非常困难的，时间问题或消息丢失都可能导致这种情况，因此广播协议并不能基于以上假设。为了解决这个问题，Zab协议提供了以下保障：

- 一个被选举的群首确保在提交完所有之前的时间戳内需要提交的事务，之后才开始广播新的事务。

·在任何时间点，都不会出现两个被仲裁支持的群首。

为了实现第一个需求，群首并不会马上处于活动状态，直到确保仲裁数量的服务器认可这个群首新的时间戳值。一个时间戳的最初状态必须包含所有的之前已经提交的事务，或者某些已经被其他服务器接受，但尚未提交完成的事务。这一点非常重要，在群首进行时间戳 e 的任何新的提案前，必须保证自时间戳开始值到时间戳 $e-1$ 内的所有提案被提交。如果一个提案消息处于时间戳 $e' < e$ ，在群首处理时间戳 e 的第一个提案消息前没有提交之前的这个提案，那么旧的提案将永远不会被提交。

对于第二个需求有些棘手，因为我们并不能完全阻止两个群首独立地运行。假如一个群首 l 管理并广播事务，在此时，仲裁数量的服务器 Q 判断群首 l 已经退出，并开始选举了一个新的群首 l' ，我们假设在仲裁机构 Q 放弃群首 l 时有一个事务 T 正在广播，而且仲裁机构 Q 的一个严格的子集记录了这个事务 T ，在群首 l' 被选举完成后，在仲裁机构 Q 之外服务器也记录了这个事务 T ，为事务 T 形成一个仲裁数量，在这种情况下，事务 T 在群首 l' 被选举后会进行提交。不用担心这种情况，这并不是个bug，Zab协议保证 T 作为事务的一部分被群首 l' 提交，确保群首 l' 的仲裁数量的支持者中至少有一个追随者确认了该事务 T ，其中的关键点在于群首 l' 和 l 在同一时刻并未获得足够的仲裁数量的支持者。

图9-5说明了这一场景，在图中，群首l为服务器s₅，l'为服务器s₃，仲裁机构由s₁到s₃组成，事务T的zxid为（1，1）。在收到第二个确认消息之后，服务器s₅成功向服务器s₄发送了提交消息来通知提交事务。其他服务器因追随服务器s₃忽略了服务器s₅的消息，注意服务器s₃所了解的zxid为（1，1），因此它知道获得管理权后的事务点。

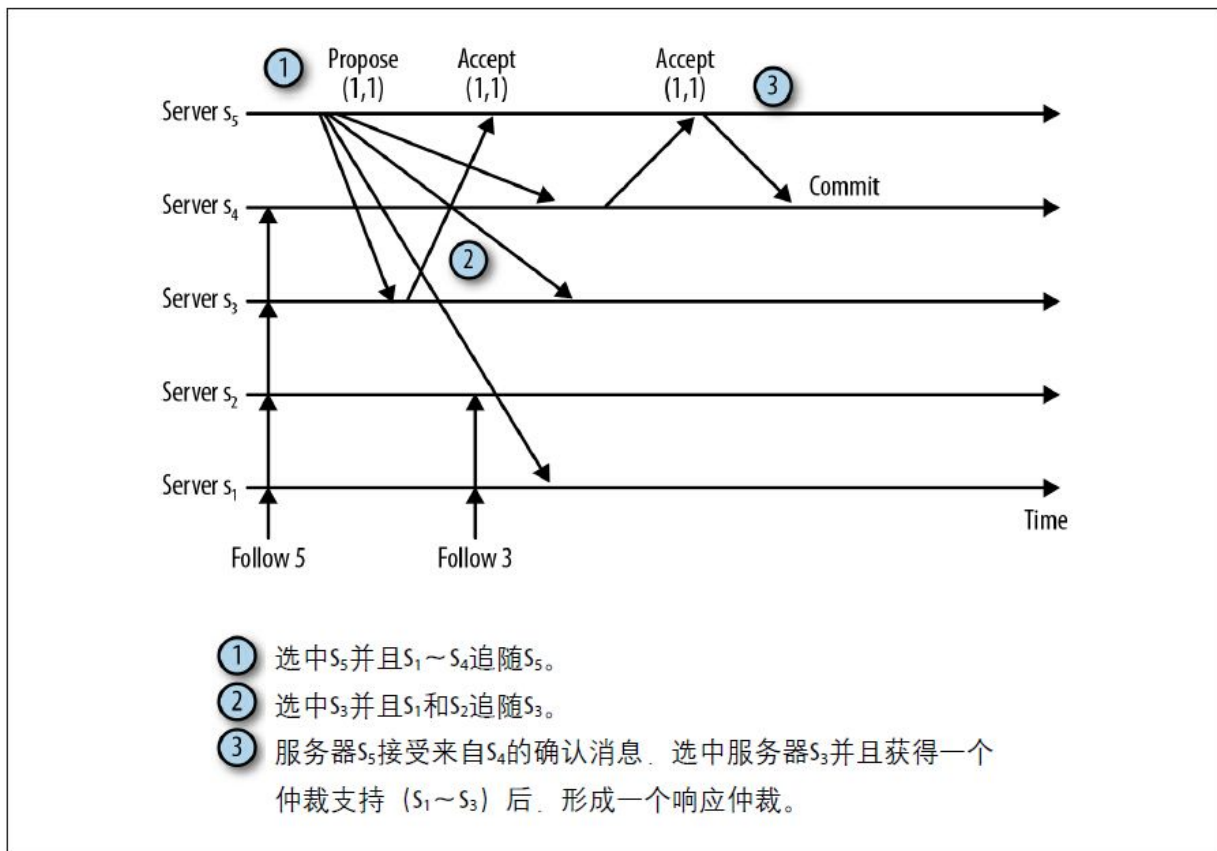


图9-5：群首发生重叠的情况

之前我们提到Zab保证新群首l'不会缺失（1，1），现在我们来查看其中的细节。在新群首l'生效前，它必须学习旧的仲裁数量服务器之前接受的所有提议，并且保证这些服务器不会继续接受来自旧群首的

提议。此时，如果群首l还能继续提交提议，比如（1， 1），这条提议必须已经被一个以上的认可了新群首的仲裁数量服务器所接受。我们知道仲裁数量必须在一台以上的服务器之上有所重叠，这样群首l用来提交的仲裁数量和新群首l使用的仲裁数量必定在一台以上的服务器上是一致的。因此，l'将（1， 1）加入自身的状态并传播给其跟随者。

在群首选举时，我们选择zxid最大的服务器作为群首。这使得ZooKeeper不需要将提议从追随者传到群首，而只需要将状态从群首传播到追随者。假设有一个追随者接受了一条群首没有接受的提议。群首必须确保在和其他追随者同步之前已经收到并接受了这条提议。但是，如果我们选择zxid最大的服务器，我们将可以完完全全跳过这一步，可以直接发送更新到追随者。

在时间戳发生转换时，Zookeeper使用两种不同的方式来更新追随者来优化这个过程。如果追随者滞后于群首不多，群首只需要发送缺失的事务点。因为追随者按照严格的顺序接收事务点，这些缺失的事务点永远是最接近的。这种更新在代码中被称之为DIFF。如果追随者滞后很久，ZooKeeper将发送在代码中被称为SNAP的完整快照。因为发送完整的快照会增大系统恢复的延时，发送缺失的事务点是更优的选择。可是当追随者滞后太远的情况下，我们只能选择发送完整快照。

群首发送给追随者的DIFF对应于已经存在于事务日志中的提议，而SNAP对应于群首拥有的最新有效快照。我们将稍后在本章中讨论这

两种保存在磁盘上的文件。

注意：深入代码

这里我们给出一点代码指导。大部分Zab的代码存在于Leader、LearnerHandler和Follower。Leader和LearnerHandler的实例由群首服务器执行，而Follower的实例由追随者执行。Leader.lead和Follower.followLeader是两个重要的方法，他们在服务器在QuorumPeer中从LOOKING转换到LEADING或者FOLLOWING时得到调用。

如果你对DIFF和SNAP的区别感兴趣，可以查看LearnerHandler.run的代码，其中包含了使用DIFF时如何决定发送哪条提议，以及关于如何持久化和发送快照的细节。

9.4 观察者

至此，我们已经介绍了群首和追随者。还有一类我们没有介绍的服务器：观察者。观察者和追随者之间有一些共同点。具体说来，他们提交来自群首的提议。不同于追随者的是，观察者不参与我们之前介绍过的选举过程。他们仅仅学习经由INFORM消息提交的提议。由于群首将状态变化发送给追随者和观察者，这两种服务器也都被称为学习者。

注意：深入INFORM消息

因为观察者不参与决定提议接受与否的投票，群首不需要发送提议到观察者，群首发送给追随者的提交消息只包含zxid而不包含提议本身。因此，仅仅发送提交消息给观察者并不能使其实施提议。这是我们使用INFORM消息的原因。INFORM消息本质上是包含了正在被提交的提议信息的提交消息。

简单来说，追随者接受两种消息而观察者只接受一种消息。追随者从一次广播中获取提议的内容，并从接下来的一条提交消息中获取zxid。相比之下，观察者只获取一条包含已提交提议的内容的INFORM消息。

参与决定那条提议被提交的投票的服务器被称为**PARTICIPANT**服务器。一个**PARTICIPANT**服务器可以是群首也可以是追随者。而观察者则被称为**OBSERVER**服务器。

引入观察者的一个主要原因是提高读请求的可扩展性。通过加入多个观察者，我们可以在不牺牲写操作的吞吐率的前提下服务更多的读操作。写操作的吞吐率取决于仲裁数量的大小。如果我们加入更多的参与投票的服务器，我们将需要更大的仲裁数量，而这将减少写操作的吞吐率。增加观察者也不是完全没有开销的。每一个新加入的观察者将对应于每一个已提交事务点引入的一条额外消息。然而，这个开销相对于增加参与投票的服务器来说小很多。

采用观察者的另外一个原因是进行跨多个数据中心的部署。由于数据中心之间的网络链接延时，将服务器分散于多个数据中心将明显地降低系统的速度。引入观察者后，更新请求能够先以高吞吐率和低延迟的方式在一个数据中心内执行，接下来再传播到异地的其他数据中心得到执行。值得注意的是，观察者并不能消除数据中心之间的网络消息，因为观察者必须转发更新请求给群首并且处理**INFORM**消息。不同的是，当参与的服务器处于同一个数据中心时，观察者保证提交更新必需的消息在数据中心内部得到交换。

9.5 服务器的构成

群首、追随者和观察者根本上都是服务器。我们在实现服务器时使用的主要抽象概念是请求处理器。请求处理器是对处理流水线上不同阶段的抽象。每一个服务器实现了一个请求处理器的序列。我们可以把一个处理器想象成添加到请求处理的一个元素。一条请求经过服务器流水线上所有处理器的处理后被称为得到完全处理。

注意：请求处理器

ZooKeeper代码里有一个叫RequestProcessor的接口。这个接口的主要方法是processRequest，它接受一个Request参数。在一条请求处理器的流水线上，对相邻处理器的请求的处理通常通过队列现实解耦合。当一个处理器有一条请求需要下一个处理器进行处理时，它将这条请求加入队列。然后，它将处于等待状态直到下一个处理器处理完此消息。

9.5.1 独立服务器

Zookeeper中最简单的流水线是独立服务器（ZeeKeeperServer类）。图9-6描述了此类服务器的流水线。它包含三种请求处理器：PrepRequestProcessor、SyncRequestProcessor和FinalRequestProcessor。

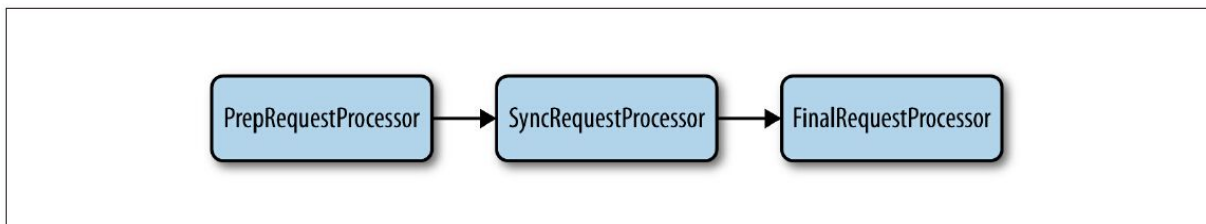


图9-6：一个独立服务器的流水线

PrepRequestProcessor接受客户端的请求并执行这个请求，处理结果则是生成一个事务。我们知道事务是执行一个操作的结果，该操作会反映到**ZooKeeper**的数据树上。事务信息将会以头部记录和事务记录的方式添加到**Request**对象中。同时还要注意，只有改变**ZooKeeper**状态的操作才会产生事务，对于读操作并不会产生任何事务。因此，对于读请求的**Request**对象中，事务的成员属性的引用值则为**null**。

下一个请求处理器为**SyncRequestProcessor**。**SyncRequestProcessor**负责将事务持久化到磁盘上。实际上就是将事务数据按顺序追加到事务日志中，并生成快照数据。对于硬盘状态的更多细节信息，我们将在本章下一节进行讨论。

下一个处理器也是最后一个为**FinalRequestProcessor**。如果**Request**对象包含事务数据，该处理器将会接受对**ZooKeeper**数据树的修改，否则，该处理器会从数据树中读取数据并返回给客户端。

9.5.2 群首服务器

当我们切换到仲裁模式时，服务器的流水线则有一些变化，首先我们介绍群首的操作流水线（类LeaderZooKeeper），如图9-7所示。

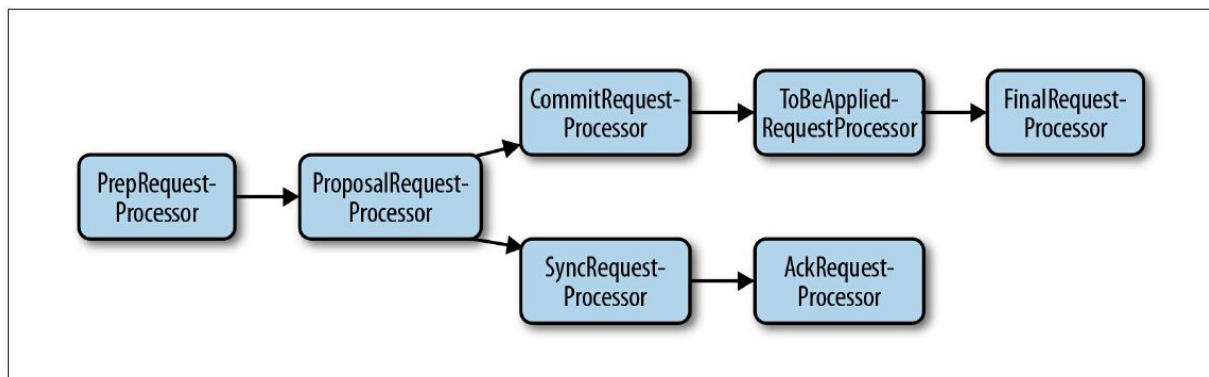


图9-7：群首服务器的流水线

第一个处理器同样是PrepRequestProcessor，而之后的处理器则为ProposalRequestProcessor。该处理器会准备一个提议，并将该提议发送给跟随者。ProposalRequestProcessor将会把所有请求都转发给CommitRequestProcessor，而且，对于写操作请求，还会将请求转发给SyncRequestProcessor处理器。

SyncRequestProcessor处理器所执行的操作与独立服务器中的一样，即持久化事务到磁盘上。执行完之后会触发AckRequestProcessor处理器，这个处理器是一个简单请求处理器，它仅仅生成确认消息并返回给自己。我们之前曾提到过，在仲裁模式下，群首需要收到每个服务器的确认消息，也包括群首自己，而AckRequestProcessor处理器就负责这个。

在ProposalRequestProcessor处理器之后的处理器为CommitRequestProcessor。CommitRequestProcessor会将收到足够多的确认消息的提议进行提交。实际上，确认消息是由Leader类处理的（Leader.processAck（）方法），这个方法会将提交的请求加入到CommitRequestProcessor类中的一个队列中。这个队列会由请求处理器线程进行处理。

下一个处理器也是最后一个为FinalRequestProcessor处理器，它的作用与独立服务器一样。FinalRequestProcessor处理更新类型的请求，并执行读取请求。在FinalRequestProcessor处理器之前还有一个简单的请求处理器，这个处理器会从提议列表中删除那些待接受的提议，这个处理器的名字叫ToBeAppliedRequestProcessor。待接受请求列表包括那些已经被仲裁法定人数所确认的请求，并等待被执行。群首使用这个列表与追随者之间进行同步，并将收到确认消息的请求加入到这个列表中。之后ToBeAppliedRequestProcessor处理器就会在FinalRequestProcessor处理器执行后删除这个列表中的元素。

注意，只有更新请求才会加入到待接受请求列表中，然后由ToBeAppliedRequest-Processor处理器从该列表移除。ToBeAppliedRequestProcessor处理器并不会对读取请求进行任何额外的处理操作，而是由FinalRequestProcessor处理器进行操作。

9.5.3 追随者和观察者服务器

现在我们来讨论追随者（`FollowerRequestProcessor`类），图9-8展示了一个追随者服务器中会用到的请求处理器。我们注意到图中并不是一个单一序列的处理器，而且输入也有不同形式：客户端请求、提议、提交事务。我们通过箭头来标识追随者处理的不同路径。

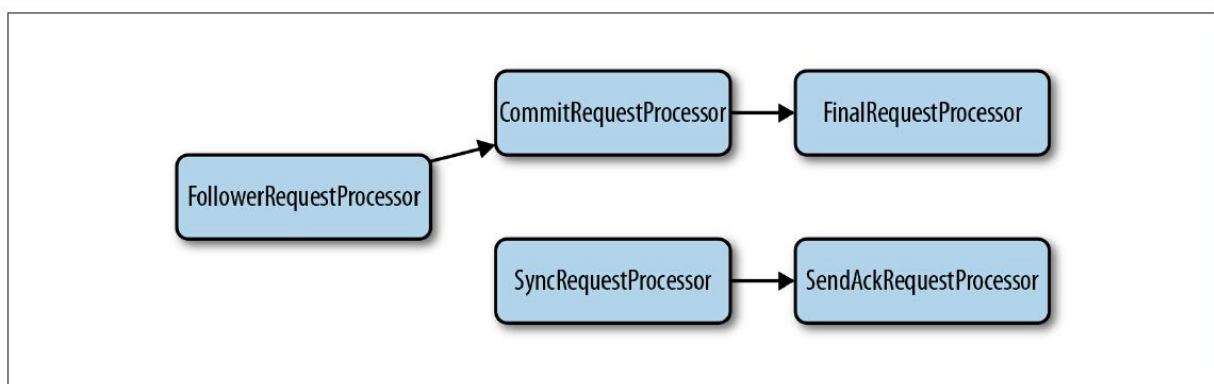


图9-8：追随者服务器的流水线

我们首先从`FollowerRequestProcessor`处理器开始，该处理器接收并处理客户端请求。`FollowerRequestProcessor`处理器之后转发请求给`CommitRequestProcessor`，同时也会转发写请求到群首服务器。`CommitRequestProcessor`会直接转发读取请求到`FinalRequestProcessor`处理器，而且对于写请求，`CommitRequestProcessor`在转发给`FinalRequestProcessor`处理器之前会等待提交事务。

当群首接收到一个新的写请求操作时，直接地或通过其他追随者服务器来生成一个提议，之后转发到追随者服务器。当收到一个提

议，追随者服务器会发送这个提议到SyncRequestProcessor处理器，SendRequestProcessor会向群首发送确认消息。当群首服务器接收到足够确认消息来提交这个提议时，群首就会发送提交事务消息给追随者（同时也会发送INFORM消息给观察者服务器）。当接收到提交事务消息时，追随者就通过CommitRequestProcessor处理器进行处理。

为了保证执行的顺序，CommitRequestProcessor处理器会在收到一个写请求处理器时暂停后续的请求处理。这就意味着，在一个写请求之后接收到的任何读取请求都将被阻塞，直到读取请求转给CommitRequestProcessor处理器。通过等待的方式，请求可以被保证按照接收的顺序来被执行。

对于观察者服务器的请求流水线（ObserverZooKeeperServer类）与追随者服务器的流水线非常相似。但是因为观察者服务器不需要确认提议消息，因此观察者服务器并不需要发送确认消息给群首服务器，也不用持久化事务到硬盘。对于观察者服务器是否需要持久化事务到硬盘，以便加速观察者服务器的恢复速度，这样的讨论正在进行中，因此对于以后的ZooKeeper版本也会有这一个功能。

9.6 本地存储

我们之前已经提到过事务日志和快照，SyncRequestProcessor处理器就是用于在处理提议是写入这些日志和快照。我们将会在本节详细讨论这些。

9.6.1 日志和磁盘的使用

我们之前说过服务器通过事务日志来持久化事务。在接受一个提议时，一个服务器（追随者或群首服务器）就会将提议的事务持久化到事物日志中，该事务日志保存在服务器的本地磁盘中，而事务将会按照顺序追加其后。服务器会时不时地滚动日志，即关闭当前文件并打开一个新的文件。

因为写事务日志是写请求操作的关键路径，因此ZooKeeper必须有效处理写日志问题。一般情况下追加文件到磁盘都会有效完成，但还有一些情况可以使ZooKeeper运行的更快，组提交和补白。组提交

（Group Commits）是指在一次磁盘写入时追加多个事务。这将使持久化多个事物只需要一次磁道寻址的开销。

关于持久化事务到磁盘，还有一个重要说明：现代操作系统通常会缓存脏页（Dirty Page），并将它们异步写入磁盘介质。然而，我们

需要在继续之前，确保事务已经被持久化。因此我们需要冲刷

(Flush) 事务到磁盘介质。冲刷在这里就是指我们告诉操作系统将脏页写入磁盘，并在操作完成后返回。因为我们在SyncRequestProcessor处理器中持久化事务，所以这个处理器同时也会负责冲刷。在SyncRequestProcessor处理器中当需要冲刷事务到磁盘时，事实上我们是冲刷的是所有队列中的事务，以实现组提交的优化。如果队列中只有一个事务，这个处理器依然会执行冲刷。该处理器并不会等待更多的事务进入队列，因为这样做会增加执行操作的延时。代码参考可以查看SyncRequestProcessor.run () 方法。

注意：磁盘写缓存

服务器只有在强制将事务写入事务日志之后才确认对应的提议。更准确一点，服务器调用ZKDatabase的commit方法，这个方法最终会调用FileChannel.force。这样，服务器保证在确认事务之前已经将它持久化到磁盘中。不过，有一个需要注意的地方，现代的磁盘一般有一个缓存用于保存将要写到磁盘的数据。如果写缓存开启，force调用在返回后并不能保证数据已经写入介质中。实际上，它可能还在写缓存中。为了保证在FileChannel.force () 方法返回后，写入的数据已经在介质上，磁盘写缓存必须关闭。不同的操作系统有不同的关闭方式。

补白 (padding) 是指在文件中预分配磁盘存储块。这样做，对于涉及存储块分配的文件系统元数据的更新，就不会显著影响文件的顺

序写入操作。假如需要高速向日志中追加事务，而文件中并没有原先分配存储块，那么无论何时在写入操作到达文件的结尾，文件系统都需要分配一个新存储块。而通过补白至少可以减少两次额外的磁盘寻址开销：一次是更新元数据；另一次是返回文件。

为了避免受到系统中其他写操作的干扰，我们强烈推荐你将事务日志写入到一个独立磁盘，将第二块磁盘用于操作系统文件和快照文件。

9.6.2 快照

快照是ZooKeeper数据树的拷贝副本，每一个服务器会经常以序列化整个数据树的方式来提取快照，并将这个提取的快照保存到文件中。服务器在进行快照时不需要进行协作，也不需要暂停处理请求。因为服务器在进行快照时还会继续处理请求，所以当快照完成时，数据树可能又发生了变化，我们称这样的快照是模糊的（fuzzy），因为它们不能反映出在任意给点的时间点数据树的准确状态。

举个例子说明一下。一个数据树中只有2个znode节点：/z和/z'。一开始，两个znode节点的数据都是1。现在有以下操作步骤：

- 1.开始一个快照。

2.序列化并将/z=1到快照。

3.使/z的数据为2（事务T）。

4.使/z'的数据为2（事务T'）。

5.序列化并将/z'=2写入到快照。

这个快照包含了/z=1和/z'=2。然而，数据树中这两个znode节点在任意的时间点上都不是这个值。这并不是问题，因为服务器会重播（replay）事务。每一个快照文件都会以快照开始时最后一个被提交的事务作为标记（tag），我们将这个时间戳记为TS。如果服务器最后加载快照，它会重播在TS之后的所有事务日志中的事务。在这个例子中，它们就是T和T'。在快照的基础上重放T和T'后，服务器最终得到/z=2和/z'=2，即一个合理的状态。

接下来我们还需要考虑一个重要的问题，就是再次执行事务T是否会有问题，因为这个事务在开始快照开始之后已经被接受，而结果也被快照中保存下来。就像我们之前所说的，事务是幂等的（idempotent），所以即使我们按照相同的顺序再次执行相同的事务，也会得到相同的结果，即便其结果已经保存到快照中。

为了理解这个过程，假设重复执行一个已经被执行过的事务。如上例中所描述，一个操作设置某个znode节点的数据为一个特定的值，

这个值并不依赖于任何其他东西，我们无条件（**unconditionally**）地设置/z'的值（**setData**请求中的版本号为-1），重新执行操作成功，但因为我们递增了两次，所以最后我们以错误的版本号结束。如以下方式就会导致问题出现，假设有如下3个操作并成功执行：

```
setData /z', 2, -1  
setData /z', 3, 2  
setData /a, 0, -1
```

第一个**setData**操作跟我们之前描述的一样，而后我们又加上了2个**setData**操作，以此来展示在重放中第二个操作因为错误的版本号而未能成功的情况。假设这3个操作在提交时被正确执行。此时如果服务器加载最新的快照，即该快照已包含第一个**setData**操作。服务器仍然会重放第一个**setData**操作，因为快照被一个更早的zxid所标记。因为重新执行了第一个**setData**操作。而第二个**setData**操作的版本号又与期望不符，那么这个操作将无法完成。而第三个**setData**操作可以正常完成，因为它也是无条件的。

在加载完快照并重放日志后，此时服务器的状态是不正确的，因为它没有包括第二个**setData**请求。这个操作违反了持久性和正确性，以及请求的序列应该是无缺口（**no gap**）的属性。

这个重放请求的问题可以通过把事务转换为群首服务器所生成的**state delta**来解决。当群首服务器为一个请求产生事务时，作为事务生

成的一部分，包括了一些在这个请求中**znode**节点或它的数据变化的值（**delta**值），并指定一个特定的版本号。最后重新执行一个事务就不会导致不一致的版本号。

9.7 服务器与会话

会话（Session）是Zookeeper的一个重要的抽象。保证请求有序、临时znode节点、监事点都与会话密切相关。因此会话的跟踪机制对ZooKeeper来说也非常重要。

ZooKeeper服务器的一个重要任务就是跟踪并维护这些会话。在独立模式下，单个服务器会跟踪所有的会话，而在仲裁模式下则由群首服务器来跟踪和维护。群首服务器和独立模式的服务器实际上运行相同的会话跟踪器（参考SessionTracker类和SessionTrackerImpl类）。而追随者服务器仅仅是简单地把客户端连接的会话信息转发给群首服务器（参考LearnerSessionTracker类）。

为了保证会话的存活，服务器需要接收会话的心跳信息。心跳的形式可以是一个新的请求或者显式的ping消息（参考LearnerHandler.run（））。两种情况下，服务器通过更新会话的过期时间来触发（touch）会话活跃（参考SessionTrackerImpl.touchSession（）方法）。在仲裁模式下，群首服务器发送一个PING消息给它的追随者们，追随者们返回自从最新一次PING消息之后的一个session列表。群首服务器每半个tick（参考10.1.1节的介绍）就会发送一个ping

消息给追随者们。所以，如果一个tick被设置成2秒，那么群首服务器就会每一秒发送一个ping消息。

对于管理会话的过期有两个重要的要点。一个称为过期队列（expiry queue）的数据结构（参考ExpiryQueue类），用于维护会话的过期。这个数据结构使用bucket来维护会话，每一个bucket对应一个某时间范围内过期的会话，群首服务器每次会让一个bucket的会话过期。为了确定哪一个bucket的会话过期，如果有的话，当下一个底限到来时，一个线程会检查这个expiry queue来找出要过期的bucket。这个线程在底限时间到来之前处于睡眠状态，当它被唤醒时，它会取出过期队列的一批session，让它们过期。当然取出的这批数据也可能是空的。

为了维护这些bucket，群首服务器把时间分成一些片段，以expirationInterval为单位进行分割，并把每个会话分配到它的过期时间对应的bucket里，其功能就是有效地计算出一个会话的过期时间，以向上取正的方式获得具体时间间隔。更具体来说，就是对下面的表达式进行计算，当会话的过期时间更新时，根据结果来决定它属于哪一个bucket。

$$(\text{expirationTime} / \text{expirationInterval} + 1) * \text{expirationInterval}$$

举例说明，比如`expirationInterval`为2，会话的超时时间为10。那么这个会话分配到`bucket`为12（ $(10/2+1)*2$ 的结果）。注意当我们触发（`touch`）这个会话时`expirationTime`会增加，所以随后我们需要根据之后的计算会话移动到其他的`bucket`中。

使用`bucket`的模式来管理的一个主要原因是为了减少让会话过期这项工作的系统开销。在一个`ZooKeeper`的部署环境中，可能其客户端就有数千个，因此也就有数千个会话。在这种场景下要细粒度地检查会话过期是不合适的。如果`expirationInterval`短的话，那么`ZooKeeper`就会以这种细粒度的方式完成检查。目前`expirationInterval`是一个`tick`，通常以秒为单位。

9.8 服务器与监视点

监视点（参考2.1.3节）是由读取操作所设置的一次性触发器，每个监视点由一个特定操作来触发。为了在服务端管理监视点，ZooKeeper的服务端实现了监视点管理器（`watch manager`）。一个`WatchManager`类的实例负责管理当前已被注册的监视点列表，并负责触发它们。所有类型的服务器（包括独立服务器，群首服务器，追随者服务器和观察者服务器）都使用同样的方式处理监视点。

`DataTree`类中持有一个监视点管理器来负责子节点监控和数据的监控，对于这两类监控，请参考4.2节，当处理一个设置监视点的读请求时，该类就会把这个监视点加入`manager`的监视点列表。类似的，当处理一个事务时，该类也会查找是否需要触发相应的监视点。如果发现有监视点需要触发，该类就会调用`manager`的触发方法。添加一个监视点和触发一个监视点都会以一个`read`请求或者`FinalRequestProcessor`类的一个事务开始。

在服务端触发了一个监视点，最终会传播到客户端。负责处理传播的为服务端的`cnxn`对象（参见`ServerCnxn`类），此对象表示客户端和服务端的连接并实现了`Watcher`接口。`Watch.process`方法序列化了监视点事件为一定格式，以便用于网络传送。ZooKeeper客户端接收序列

化的监视点事件，并将其反序列化为监视点事件的对象，并传递给应用程序。

监视点只会保存在内存，而不会持久化到硬盘。当客户端与服务端的连接断开时，它的所有监视点会从内存中清除。因为客户端库也会维护一份监视点的数据，在重连之后监视点数据会再次被同步到服务端。

9.9 客户端

在客户端库中有2个主要的类： `ZooKeeper`和`ClientCnxn`。

`ZooKeeper`类实现了大部分API，写客户端应用程序时必须实例化这个类来建立一个会话。一旦建立起一个会话，`ZooKeeper`就会使用一个会话标识符来关联这个会话。这个会话标识符实际上是由服务端所生成的（参考`SessionTrackerImpl`类）。

`ClientCnxn`类管理连接到server的Socket连接。该类维护了一个可连接的`ZooKeeper`的服务器列表，并当连接断掉的时候无缝地切换到其他的服务器。当重连到一个其他的服务器时会使用同一个会话（如果没有过期的话），客户端也会重置所有的监视点到刚连接的服务器上（参考`ClientCnxn.SendThread.primeConnection()`）。重置默认是开启的，可以通过设置`disableAutoWatchReset`来禁用。

9.10 序列化

对于网络传输和磁盘保存的序列化消息和事务，**ZooKeeper**使用了**Hadoop**中的**Jute**来做序列化。如今，该库以独立包的方式被引入，在**ZooKeeper**代码库中，`org.apache.jute`就是**Jute**库（**ZooKeeper**的开发团队早就讨论过要替换**Jute**，但至今没找到合适的方案，它工作得很好，还没有必要替换它）。

对于**Jute**最主要的定义文件为`zookeeper.jute`。它包含了所有的消息定义和文件记录。下面是一个**Jute**定义的例子：

```
module org.apache.zookeeper.txn {  
    ...  
    class CreateTxn {  
        ustring path;  
        buffer data;  
        vector<org.apache.zookeeper.data.ACL> acl;  
        boolean ephemeral;  
        int parentCVersion;  
    }  
    ...  
}
```

这个例子定义模块，该模块包括一个**create**事务的定义。同时。这个模块映射到了一个**ZooKeeper**的包中。

9.11 小结

本章讨论了ZooKeeper核心机制问题。群首竞选机制是可用性的关键因素，没有这个机制，ZooKeeper套件将无法保持可靠性。拥有群首是必要但非充分条件，ZooKeeper还需要Zab协议来传播状态的更新等，即使某些服务器可能发生崩溃，也能保证状态的一致性。

我们又回顾了多种服务器类型：独立服务器、群首服务器、追随者服务器和观察者服务器。这些服务器之间因运转的机制及执行的协议的不同而不同。在不同的部署场景中，各个服务器可以发挥不同的作用，比如增加观察者服务器可以提供更高的读吞吐量，而且还不会影响写吞吐量。不过，增加观察者服务器并不会增加整个系统的高可用性。

在ZooKeeper内部，我们实现了一系列机制和数据结构。我们在本章中专注于会话与监视点的实现，这个也是在实现ZooKeeper应用时所涉及的重要概念。

虽然我们在本章中提供了代码的相关线索，但我们并没有提供源代码的详尽视图。我们强烈建议读者自行下载一份源代码，以本章所提供的线索为开端，独立分析和思考。

第10章 运行ZooKeeper

ZooKeeper的设计不仅为开发人员提供了很好的构建模块，也为运维人员以提供了很多方便。分布式系统越来越庞大，管理运维越来越困难，更加健壮的运维实践越来越重要。我们期望Zookeeper作为一个标准的分布式系统组件，可以让运维团队更方便地学习和管理。我们在之前的章节中已经看到启动ZooKeeper非常简单，但是在运行ZooKeeper服务时还有很多选项需要注意。本章的主要介绍运行ZooKeeper时可用的管理工具和方法。

为了保证Zookeeper服务的功能正常，必须保证配置正确。ZooKeeper作为分布式计算的基础建立在需要条件处理的情况下，所有的Zookeeper投票服务器必须拥有相同的配置，在我们的经验中，配置错误或不一致是运营过程中最主要的问题。

我们先看一个使用ZooKeeper早期出现的一个问题的简单例子，一个团队的早期人员通过ZooKeeper编写了他们的应用，经过完全测试后，部署到生产环境中，在早期ZooKeeper很容易使用和部署，所以这个团队在生产环境部署了他们的ZooKeeper服务和应用程序，但并未告知运维人员。

不久之后，随着生产流量开始进入，问题也开始显现出来。我们接连收到运营团队的电话，告知系统出现了问题，运维人员不断地质问我们这个系统方案在部署到生产环境前是否经过完全测试，我们也在不断地重复回答他，已经完成了完全测试。在经过各种场景的拼凑整理，运维人员意识到所遇到的问题应该是脑裂问题，最后，我们要求他发送一下他们的配置文件，当我们看到这个配置文件，我们很清楚问题出在哪里：我们使用了独立模式的ZooKeeper进行测试，但在部署到生产环境时，为了可以容忍某个服务器的故障部署了三个服务器。遗憾的是，运维人员忘记修改配置文件，因此他们将整个应用置于三个独立模式的服务器之中。

客户端将这三个服务器当成集群中的一个部分，但服务器之间却独立运行着，因此三组不同的客户端在系统中看到了三个不同且冲突的视图，表面看起来似乎工作正常，但在后台却一片混乱。

通过这个例子来说明ZooKeeper的配置是很重要的事情，你需要了解一些基本概念，其实这些并不难或复杂，关键是知道选项开关的位置和具体含义，本章中就会介绍这些配置选项的含义。

10.1 配置ZooKeeper服务器

在这一节，我们将会看到那些负责ZooKeeper服务器运转的选项，之前我们已经看到了部分选项，但其实还有很多选项没有介绍过，这些选项均有默认值，符合最常见情况下运转，但这些选项常常需要修改。ZooKeeper的设计在于方便使用和运维，我们的设计是成功的，又使人们还不太明白自己的配置就可以动手并运行起来。人们很容易使用最简配置开始运行ZooKeeper，但如果你多花一些时间学习一下不同的配置选项，你会发现你还可以获得更高的性能，而且更容易诊断问题所在。

在本节中我们将介绍每一个配置选项参数，以及这些参数的具体含义，并且介绍为什么你需要使用这个参数。本节内容也许会让你觉得很枯燥，如果你想看一些更有趣的知识，可以先跳到下一节中。如果跳过本节，最好回头在某种程度上沉下心看一看，熟悉一下不同的选项含义，这些选项参数会对你的ZooKeeper服务在稳定性和性能上形成巨大差异。

ZooKeeper服务器在启动时从一个名为`zoo.cfg`的配置文件读取所有选项，多个服务器如果角色相似，同时基本配置信息一样，就可以共享一个文件。`data`目录下的`myid`文件用于区分各个服务器，对每个服

务器来说，**data**目录必须是唯一的，因此这个目录可以更加方便地保存一些差异化文件。服务器ID将**myid**文件作为一个索引引入到配置文件中，一个特定的**ZooKeeper**服务器可以知道如何配置自己参数。当然，如果服务器具有不同的配置参数（例如，事务日志保存在不同的地方），每个服务器就需要使用自己唯一的配置文件。

配置参数常常通过配置文件的方式进行设置，本节后续部分，通过列表方式列出了这些参数。很多参数也可以通过**Java**的系统属性传递，其形式通常为**zookeeper.propertyName**，在启动服务器时，通过**-D**选项设置这些属性。不过，系统属性所对应的一个特定参数对服务来说是插入的配置，配置文件中的配置参数优先于系统属性中的配置。

10.1.1 基本配置

某些配置参数并没有默认值，所以每个部署应用中必须配置：

clientPort

客户端所连接的服务器所监听的**TCP**端口，默认情况下，服务端会监听在所有的网络连接接口的这个端口上，除非设置了**clientPortAddress**参数。客户端端口可以设置为任何值，不同的服务器也可以监听在不同的端口上。默认端口号为**2181**。

dataDir和dataLogDir

dataDir用于配置内存数据库保存的模糊快照的目录，如果某个服务器为集群中的一台，**id**文件也保存在该目录下。

dataDir并不需要配置到一个专用存储设备上，快照将会以后台线程的方式写入，且并不会锁定数据库，而且快照的写入方式并不是同步方式，直到写完整快照为止。

事务日志对该目录所处的存储设备上的其他活动更加敏感，服务端会尝试进行顺序写入事务日志，以为服务端在确认一个事务前必须将数据同步到存储中，该设备的其他活动（尤其是快照的写入）可能导致同步时磁盘过于忙碌，从而影响写入的吞吐能力。因此，最佳实践是使用专用的日志存储设备，将**dataLogDir**的目录配置指向该设备。

tickTime

tick的时长单位为毫秒，**tick**为ZooKeeper使用的基本的时间度量单位，在9.7节已经介绍过，该值还决定了会话超时的存储器大小。

Zookeeper集群中使用的超时时间单位通过**tickTime**指定，也就是说，实际上**tickTime**设置了超时时间的下限值，因为最小的超时时间为一个**tick**时间，客户端最小会话超时事件为两个**tick**时间。

`tickTime`的默认值为3000毫秒，更低的`tickTime`值可以更快地发现超时问题，但也会导致更高的网络流量（心跳消息）和更高CPU使用率（会话存储器的处理）。

10.1.2 存储配置

该小节覆盖了一些更高级的配置参数，这些参数不仅适用于独立模式的服务，也适用于集群模式下的配置，这些参数不设置的话并不会影响ZooKeeper的功能，但有些参数（例如`dataLogDir`）最好还是配置：

`preAllocSize`

用于设置预分配的事务日志文件（`zookeeper.preAllocSize`）的大小值，以KB为单位。

当写入事务日志文件时，服务端每次会分配`preAllocSize`值的KB的存储大小，通过这种方式可以分摊文件系统将磁盘分配存储空间和更新元数据的开销，更重要的是，该方式也减少了文件寻址操作的次数。

默认情况下`preAllocSize`的值为64MB，缩小该值的一个原因是事务日志永远不会达到这么大，因为每次快照后都会重新启动一个新的

事务日志，如果每次快照之间的日志数量很小，而且每个事务本身也很小，64MB的默认值显然就太大了。例如，如果我们每1000个事务进行一次快照，每个事务的平均大小为100字节，那么100KB的preAllocSize值则更加合适。默认的preAllocSize值的设置适用于默认的snapCount值和平均事务超过512字节的情况。

snapCount

指定每次快照之间的事务数（zookeeper.snapCount）。

当Zookeeper服务器重启后需要恢复其状态，恢复时两大时间因素分别是为恢复状态而读取快照的时间以及快照启动后所发生的事务的执行时间。执行快照可以减少读入快照文件后需要应用的事务数量，但是进行快照时也会影响服务器性能，即便是通过后台线程的方式进行写入操作。

snapCount的默认值为100000，因为进行快照时会影响性能，所以集群中所有服务器最好不要在同一时间进行快照操作，只要仲裁服务器不会一同进行快照，处理时间就不会受影响，因此每次快照中实际的事务数为一个接近snapCount值的随机数。

注意，如果snapCount数已经达到，但前一个快照正在进行中，新的快照将不会开始，服务器也将继续等到下一个snapCount数量的事务后再开启一个新的快照。

`autopurge.snapRetainCount`

当进行清理数据操作时，需要保留在快照数量和对应的事务日志文件数量。

ZooKeeper将会定期对快照和事务日志进行垃圾回收操作，`autopurge.snapRetainCount`值指定了垃圾回收时需要保留的快照数，显然，并不是所有的快照都可以被删除，因为那样就不可能进行服务器的恢复操作。`autopurge.snapRetainCount`的最小值为3，也是默认值的大小。

`autopurge.purgeInterval`

对快照和日志进行垃圾回收（清理）操作的时间间隔的小时数。如果设置为一个非0的数字，`autopurge.purgeInterval`指定了垃圾回收周期的时间间隔，如果设置为0，默认情况下，垃圾回收不会自动执行，而需要通过ZooKeeper发行包中的`zkCleanup.sh`脚本手动运行。

`fsync.warningthresholdms`

触发警告的存储同步时间阈值（`fsync.warningthresholdms`），以毫秒为单位。

ZooKeeper服务器在应答变化消息前会同步变化情况到存储中。如果同步系统调用消耗了太长时间，系统性能就会受到严重影响，服务

器会跟踪同步调用的持续时间，如果超过`fsync.warningthresholdms`只就会产生一个警告消息。默认情况下，该值为1000毫秒。

`weight.x=n`

该选项常常以一组参数进行配置，该选项指定组成一个仲裁机构的某个服务器的权重为`n`，其权重`n`值指示了该服务器在进行投票时的权重值。在ZooKeeper中一些部件需要投票值，比如群首选举中和原子广播协议中。默认情况下，一个服务器的权重值为1，如果定义的一组服务器没有指定权重，所有服务器的权重值将默认分配为1。

`traceFile`

持续跟踪ZooKeeper的操作，并将操作记录到跟踪日志中，跟踪日志的文件名为`traceFile.year.month.day`。除非设置了该选项（`requestTraceFile`），否则跟踪功能将不会启用。

该选项用来提供ZooKeeper所进行的操作的详细视图。不过，要想记录这些日志，ZooKeeper服务器必须序列化操作，并将操作写入磁盘，这将争用CPU和磁盘的时间。如果你使用了该选项，请确保不要将跟踪文件放到日志文件的存储设备中。我们还需要知道，跟踪选项还可能影响系统运行，甚至可能会很难重现跟踪选项关闭时发生的问题。另外还有个有趣的问题，`traceFile`选项的Java系统属性配置中不含

有zookeeper前缀，而且系统属性的名称也与配置选项名称不同，这一点请小心。

10.1.3 网络配置

这些配置参数可以限制服务器和客户端之间的通信，超时选项也在该节进行讨论：

globalOutstandingLimit

ZooKeeper中待处理请求的最大值
(`zookeeper.globalOutstandingLimit`)。

ZooKeeper客户端提交请求比ZooKeeper服务端处理请求要快很多，服务端将会对接收到的请求队列化，最终（也许几秒之内）可能导致服务端的内存溢出。为了防止发生这个问题，ZooKeeper服务端中如果待处理请求达到`globalOutstandingLimit`值就会限制客户端的请求。但是`globalOutstandingLimit`值并不是硬限制，因为每个客户端至少有一个待处理请求，否则会导致客户端超时，因此，当达到`globalOutstandingLimit`值后，服务端还会继续接收客户端连接中的请求，条件是这个客户端在服务器中没有任何待处理的请求。

为了确定某个服务器的全局限制值，我们只是简单地将该参数值除以服务器的数量，目前还没有更智能的方式去实现全局待处理操作数量的计算，并强制采用该参数所指定的限制值，因此，该限制值为待处理请求的上限值，事实上，服务器之间完美的负载均衡解决方案还无法实现，所以某些服务器运行得稍缓慢一点，或者处于更高的负载中，即使最终没有达到全局限制值也可能被限制住吞吐量。

该参数的默认值为1000个请求，你可能并不会修改该参数值，但如果你有很多客户端发送大数据包请求可能就需要降低这个参数值，但我们在实践中还未遇到需要修改这个参数的情况。

`maxClientCnxns`

允许每个IP地址的并发socket连接的最大数量。Zookeeper通过流量控制和限制值来避免过载情况的发生。一个连接的建立所使用的资源远远高于正常操作请求所使用的资源。我们曾看到过某些错误的客户端每秒创建很多ZooKeeper连接，最后导致拒绝服务（DoS），为了解决这个问题，我们添加了这个选项，通过设置该值，可以在某个IP地址已经有maxClientCnxns个连接时拒绝该IP地址新的连接。该选项的默认值为60个并发连接。

注意，每个服务器维护着这个连接的数量，如果我们有一个5个服务器的集群，并且使用默认的并发连接数60，一个欺诈性的客户端会

随机连接到这5个不同的服务器，正常情况下，该客户端几乎可以从单个IP地址上建立300个连接，之后才会触发某个服务器的限制。

clientPortAddress

限制客户端连接到指定的接收信息的地址上。默认情况下，一个ZooKeeper服务器会监听在所有的网络接口地址上等待客户端的连接。

有些服务器配置了多个网络接口，其中一个网络接口用于内网通信，另一个网络接口用于公网通信，如果你并不希望服务器在公网接口接受客户端的连接，只需要设置clientPortAddress选项为内网接口的地址。

minSessionTimeout

最小会话超时时间，单位为毫秒。当客户端建立一个连接后就会请求一个明确的超时值，而客户端实际获得的超时值不会低于minSessionTimeout的值。

ZooKeeper开发人员很想立刻且准确地检测出客户端故障发生的情况，遗憾的是，在1.1.4节已经介绍过，系统不可能实时处理这种情况，而是通过心跳和超时来处理。超时取决于ZooKeeper客户端与服务器之间的响应能力，更重要的是两者之间的网络延时和可靠性。超时时间允许的最小值为客户端与服务器之间网络的环回时间，但偶尔还

是可能发生丢包现象，当这种情况发生时，因为重传超时导致接收响应时间的增加，并会导致接收重发包的延时。

`minSessionTimeout`的默认值为`tickTime`值的两倍。配置该参数值过低可能会导致错误的客户端故障检测，配置该参数值过高会延迟客户端故障的检测时间。

`maxSessionTimeout`

会话的最大超时时间值，单位为毫秒。当客户端建立一个连接后就会请求一个明确的超时值，而客户端实际获得的超时值不会高于`maxSessionTimeout`的值。

虽然该参数并不会影响系统的性能，但却可以限制一个客户端消耗系统资源的时间，默认情况下`maxSessionTimeout`的时间为`tickTime`的20倍。

10.1.4 集群配置

当以一个集群来构建ZooKeeper服务时，我们需要为每台服务器配置正确的时间和服务器列表信息，以便服务器之间可以互相建立连接并进行故障监测，在ZooKeeper的集群中，这些参数的配置必须一致：

`initLimit`

对于追随者最初连接到群首时的超时值，单位为tick值的倍数。

当某个追随者最初与群首建立连接时，它们之间会传输相当多的数据，尤其是追随者落后整体很多时。配置initLimit参数值取决于群首与追随者之间的网络传输速度情况，以及传输的数据量大小，如果ZooKeeper中保存的数据量特别大（即存在大量的znode节点或大数据集）或者网络非常缓慢，就需要增大initLimit值，因为该值取决于环境问题，所有没有默认值。你需要为该参数配置适当的值，以便可以传输所期望的最大快照，也许有时你需要多次传输，你可以配置initLimit值为两倍你所期望的值。如果配置initLimit值过高，那么首次连接到故障的服务器就会消耗更多的时间，同时还会消耗更多的恢复时间，因此最好在你的网络中进行追随者与群首之间的网络基准测试，以你规划所使用的数据量来测试出你所期望的时间。

syncLimit

对于追随者与群首进行sync操作时的超时值，单位为tick值的倍数。

追随者总是会稍稍落后于群首，但是如果因为服务器负载或网络问题，就会导致追随者落后群首太多，甚至需要放弃该追随者，如果群首与追随者无法进行sync操作，而且超过了syncLimit的tick时间，就会放弃该追随者。与initLimit参数类似，syncLimit也没有默认值，与

initLimit不同的是，syncLimit并不依赖于ZooKeeper中保存的数据量大小，而是依赖于网络的延迟和吞吐量。在高延迟网络环境中，发送数据和接收响应包会耗费更多时间，此时就需要调高syncLimit值。即使在相对低延迟的网络中，如果某些相对较大的事务传输给追随者需要一定的时间，你也需要提高syncLimit值。

leaderServes

配置值为“yes”或“no”标志，指示群首服务器是否为客户端提供服务（zookeeper.leaderServes）。

担任群首的ZooKeeper服务器需要做很多工作，它需要与所有追随者进行通信并会执行所有的变更操作，也就意味着群首的负载会比追随者的负载高，如果群首过载，整个系统可能都会受到影响。

该标志位如果设置为“no”就可以使群首除去服务客户端连接的负担，使群首将所有资源用于处理追随者发送给它的变更操作请求，这样可以提高系统状态变更操作的吞吐能力。换句话说，如果群首不处理任何与其直连的客户端连接，追随者就会有更多的客户端，因为连接到群首的客户端将会分散到追随者上，尤其注意在集群中服务器数量比较少的时候。默认情况下，leaderServes的值为“yes”。

server.x=[hostname]: n: n[: observer]

服务器x的配置参数。

ZooKeeper服务器需要知道它们如何通信，配置文件中该形式的配置项就指定了服务器x的配置信息，其中x为服务器的ID值（一个整数）。当一个服务器启动后，就会读取data目录下myid文件中的值，之后服务器就会使用这个值作为查找server.x项，通过该项中的数据配置服务器自己。如果需要连接到另一个服务器y，就会使用server.y项的配置信息来与这个服务器进行通信。

其中hostname为服务器在网络n中的名称，同时后面跟了两个TCP的端口号，第一个端口用于事务的发送，第二个端口用于群首选举，典型的端口号配置为2888: 3888。如果最后一个字段标记了observer属性，服务器就会进入观察者模式。

注意，所有的服务器使用相同的server.x配置信息，这一点非常重要，否则的话，因服务器之间可能无法正确建立连接而导致整个集群无法正常工作。

cnxTimeout

在群首选举打开一个新的连接的超时值（zookeeper.cnxTimeout）。

ZooKeeper服务器在进行群首选举时互相之间会建立连接，该选项值确定了一个服务器在进行重试前会等待连接成功建立的时间为多久，9.2节介绍了该超时的用途。默认的超时时间为5秒，该值足够大，也许你并不需要修改。

`electionAlg`

选举算法的配置选项。

为了整个配置的完整性，我们也列入了该选项。该选项用于选择不同的群首选举算法，但除了默认的配置外，其他算法都已经弃用了，所以你并不需要配置这个选项。

10.1.5 认证和授权选项

该节中包括认证和授权相关的选型配置。对于Kerberos相关的配置选项信息，请参考6.1.2节：

`zookeeper.DigestAuthenticationProvider.superDigest`（只适用于Java系统属性）该系统属性指定了“super”用户的密码摘要信息（该功能默认不启用），以super用户认证的客户端会跳过所有ACL检查。该系统属性的值形式为`super: encoded_digest`。为了生成加密的摘要，可以使

用org.apache.zookeeper.server.auth.DigestAuthenticationProvider工具，使用方式如下：

```
java -cp $ZK_CLASSPATH \  
    org.apache.zookeeper.server.auth.DigestAuthenticationProvider super:asdf
```

通过命令行工具生成了一个asdf这个密码的加密摘要信息：

```
super:asdf->super:T+4Qoey4ZZ8Fnni1Yl2GZtbH2W4=
```

为了在服务器启动中使用该摘要，可以通过以下命令实现：

```
export SERVER_JVMFLAGS  
SERVER_JVMFLAGS=-Dzookeeper.DigestAuthenticationProvider.superDigest=  
    super:T+4Qoey4ZZ8Fnni1Yl2GZtbH2W4=  
./bin/zkServer.sh start
```

现在，当我们通过zkCli进行连接时，可以通过以下方式：

```
[zk: localhost:2181(CONNECTED) 0] addauth digest super:asdf  
[zk: localhost:2181(CONNECTED) 1]
```

此时，你已经以super用户的身份被认证，现在不会被任何ACL所限制。

注意：不安全连接

ZooKeeper客户端与服务器之间的连接并未加密，因此不要在不可信的链接中使用super的密码，使用super密码的安全方式是在

ZooKeeper服务器本机上使用super密码运行客户端。

10.1.6 非安全配置

以下的配置选项也许会有用，但在你使用这些配置时需要小心，这些配置选项只用于非常特殊情况，大多数运维人员认为并不需要这些配置：

forceSync

通过“yes”或“no”选项可以控制是否将数据信息同步到存储设备上（`zookeeper.forceSync`）。

默认情况下，`forceSync`配置yes时，事务只有在同步到存储设备后才会被应答，同步系统调用的消耗很大，而且也是事务处理中最大的延迟原因之一。如果`forceSync`配置为no，事务会在写入到操作系统后就立刻被应答，在将事务写入磁盘之前，这些事务常常缓存于内存之中，配置`forceSync`为no可以提高性能，但代价是服务器崩溃或停电故障时可恢复性。

jute.maxbuffer（仅适用于Java系统属性）

一个请求或响应的最大值，以字节为单位。该选项只能通过Java的系统属性进行配置，并且选项名称没有zookeeper前缀。

ZooKeeper中内置了一些健康检查，其中之一就是对可传输的znode节点数据的大小的检查，ZooKeeper被设计用于保存配置数据，配置数据一般由少量的元数据信息（大约几百字节）所组成。默认情况下，一个请求或响应消息如果大于1M字节，就会被系统拒绝，你可以使用该属性来修改健康检查值，调小检查值，或者你真的确认要调大检查值。

注意：修改健康检查值

虽然通过jute.maxbuffer指定的限制值可以进行大块数据的写入操作，但获取一个znode节点的子节点，而同时该节点有很多子节点时就会出现問題。如果一个znode节点含有几十万个子节点，每个子节点的名字长度平均为10个字符，在试着返回子节点列表时就会命中默认最大缓冲大小检查，此时就会导致连接被重置。

skipACL

跳过所有ACL检查（zookeeper.skipACL）。

处理ACL检查会有一定的开销，通过该选项可以关闭ACL检查功能，这样做可以提高性能，但也会将数据完全暴露给任何一个可以连接到服务器的客户端。

readonlymode.enabled（仅适用于Java系统属性）

将该配置设置为true可以启用服务器只读模式功能，客户端可以以只读模式的请求连接服务器并读取信息（可能是已过期的信息），即使该服务器在仲裁中因分区问题而被分隔。为了启用只读模式，客户端需要配置canBeReadOnly为true。

该功能可以使客户端即使在网络分区发生时也能读取（不能写入）ZooKeeper的状态，在这种情况下，被分区而分离的客户端依然可以继续取得进展，并不需要等待分区问题被修复。特别注意，一个与集群中其他服务器失去连接ZooKeeper也许会终止以只读模式提供过期的数据服务。

10.1.7 日志

ZooKeeper采用SLF4J库（JAVA简易日志门面）作为日志的抽象层，默认使用Log4J进行实际的日志记录功能。使用两层日志抽象看起来似乎有些多余，的确是这样。本节中，我们将会简单介绍如何配置Log4J，虽然Log4J非常灵活且功能强大，但是也有点复杂。关于Log4J有专门的书籍介绍，本节中，我们只是简单介绍其基本功能。

Log4J的配置文件为log4j.properties，系统会从classpath中加载这个文件，对于Log4J比较失望的是，如果对应路径不存在log4j.properties文件，我们会看到以下输出信息：

```
log4j:WARN No appenders could be found for logger (org.apache.zookeeper.serv ...
log4j:WARN Please initialize the log4j system properly.
```

这有这些，之后所有的日志消息都会被丢弃。

一般，log4j.properties会保存到classpath中的conf目录下，让我们看一看在ZooKeeper中的log4j.properties文件的主要部分：

```
zookeeper.root.logger=INFO, CONSOLE①
```

```
zookeeper.console.threshold=INFO
zookeeper.log.dir=.
zookeeper.log.file=zookeeper.log
zookeeper.log.threshold=DEBUG
zookeeper.tracelog.dir=.
zookeeper.tracelog.file=zookeeper_trace.log
log4j.rootLogger=${zookeeper.root.logger}②
```

```
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender③
```

```
log4j.appender.CONSOLE.Threshold=${zookeeper.console.threshold}④
```

```
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout⑤
```

```
log4j.appender.CONSOLE.layout.ConversionPattern=%d{ISO8601} [myid:%X{myid}] -
...
log4j.appender.ROLLINGFILE=org.apache.log4j.RollingFileAppender⑥
```

```
log4j.appender.ROLLINGFILE.Threshold=${zookeeper.log.threshold}⑦
```

```
log4j.appender.ROLLINGFILE.File=${zookeeper.log.dir}/${zookeeper.log.file}  
log4j.appender.ROLLINGFILE.MaxFileSize=10MB  
log4j.appender.ROLLINGFILE.MaxBackupIndex=10  
log4j.appender.ROLLINGFILE.layout=org.apache.log4j.PatternLayout  
log4j.appender.ROLLINGFILE.layout.ConversionPattern=%d{ISO8601} [myid:%X{myid}] -  
...
```

①第一组配置中，所有配置项均以`zookeeper.`开头，配置了该文件的默认值，这些配置项实际上是系统属性配置，可以通过`java`命令行指定`-D`参数来覆盖JVM的配置。第一行的日志配置中，默认配置了日志消息的级别为`INFO`，即所有低于`INFO`级别的日志消息都会被丢弃，使用的`appender`为`CONSOLE`。你可以指定多个`appender`，例如，你如果想将日志信息同时输出到`CONSOLE`和`ROLLINGFILE`时，可以配置`zookeeper.root.logger`项为`INFO, CONSOLE, ROLLINGFILE`。

②`rootLogger`指定了处理所有日志消息的日志处理器，因为我们并不需要其他日志处理器。

③该行配置以`CONSOLE`名称定义了一个类，该类会处理消息的输出。在这里使用的是`ConsoleAppender`类。

④在`appender`的定义中也可以过滤消息，该行配置了这个`appender`会忽略所有低于`INFO`级别的消息，因为`zookeeper.root.logger`中定义了全局阈值为`INFO`。

⑤appender使用的布局类对输出日志在输出前进行格式化操作。我们通过布局模式定义了输出日志消息外还输出日志级别、时间、线程信息和调用位置等信息。

⑥RollingFileAppender实现了滚动日志文件的输出，而不是不断地输出到一个单个日志文件或控制台。除非ROLLINGFILE被rootLogger引用，否则该appender会被忽略。

⑦定义ROLLINGFILE的输出级别为DEBUG，因为rootLogger过滤了所有低于INFO级别的日志，所以，你如果你想看DEBUG消息，就必须将zookeeper.root.logger项的配置从INFO修改为DEBUG。

日志记录功能会影响进程的性能，特别是开启了DEBUG级别时，与此同时，DEBUG级别提供了大量有价值的信息，可以帮我们诊断问题。有一个方法可以平衡性能和开销问题，详细日志为你提供了更多细节信息，因此我们可以配置appender的日志级别为DEBUG，而rootLogger的级别为WARN，当服务器运行时，如果你需要诊断某个问题，你可以通过JMX动态调整rootLogger的级别为INFO或DEBUG，更详细地检查系统的活动情况。

10.1.8 专用资源

当你考虑在服务器上运行ZooKeeper如何配置时，服务器本身的配置也很重要。为了达到你所期望的性能，可以考虑使用专用的日志存储设备，就是说日志目录处于专属的硬盘上，没有其他进程使用该硬盘资源，甚至周期性的模糊快照也不会使用该硬盘。

10.2 配置ZooKeeper集群

关于仲裁（quorum）的概念，请参考2.2.1节，该概念深深贯穿于ZooKeeper的设计之中。在复制模式下处理请求时以及选举群首时都与仲裁的概念有关，如果ZooKeeper集群中存在法定人数的服务器已经启动，整个集群就可以继续工作。

与之相关的一个概念是观察者（observer），请参考9.4节。观察者与集群一同工作，接收客户端请求并处理服务器上的状态变更，但是群首并不会等待观察者处理请求的响应包，同时集群在进行群首选举时也不会考虑观察者的通知消息。本节我们就来讨论一下如何对仲裁和集群进行配置。

10.2.1 多数原则

当集群中拥有足够的ZooKeeper服务器来处理请求时，我们称这组服务器的集合为仲裁法定人数，我们不希望有两组不相交的服务器集合同时处理请求，否则我们会进入脑裂模式中。我们可以避免脑裂问题，通过定义仲裁法定人数的数量至少为所有服务器中的多数。

（注意，集群服务器数量的一般并不会构成多数原则，至少需要大于所有服务器一半数量来构成多数原则）

当配置多个服务器来组成ZooKeeper集群时，我们默认使用多数原则作为仲裁法定人数。ZooKeeper会自动监测是否运行于复制模式，从配置文件读取时确定是否拥有多个服务器的配置信息，并默认使用多数原则的仲裁法定人数。

10.2.2 法定人数的可配置性

我们之前提到过关于法定人数的一个重要属性是，如果一个法定人数解散了，集群中另一个法定人数形成，这两个法定人数中至少有一个服务器必须交集。多数原则的法定人数无疑满足了这一交集的属性。一般，法定人数并未限制必须满足多数原则，ZooKeeper也允许灵活的法定人数配置，这种特殊方案就是对服务器进行分组配置时，我们会将服务器分组成不相交的集合并分配服务器的权重，通过这种方案来组成法定人数，我们需要使多数组中的服务器形成多数投票原则。例如，我们有三个组，每个组中有三个服务器，每个服务器的权重值为1，在这种情况下，我们需要四个服务器来组成法定人数：某个组中的两个服务器，另一组中的两台服务器。总之，其数学逻辑归结为：如果有 G 个组，我们所需要的服务器为一个 G 组的服务器，满足 $|G| > |G|/2$ ，同时对于 G 组中的服务器集合 g ，我们还需要集合 g 中的集合 g 满足集合 g 的所有权重值之和 W' 不小于集合 g 的权重值之和（如： $W' > W/2$ ）。

通过以下配置选项可以创建一个组：

`group.x=n[: n]`

启用法定人数的分层构建方式。`x`为组的标识符，等号后面的数字对应服务器的标识符，赋值操作符右侧为冒号分隔的服务器标识符的列表。注意，组与组之间不能存在交集，所有组的并集组成了整个ZooKeeper集群，换句话说，集群中的每个服务器必须在某个组中被列出一次。

下面的示例说明了9个服务器被分为3组的情况：

```
group.1=1:2:3
group.2=4:5:6
group.3=7:8:9
```

这个例子中，每个服务器的权重都一样，为了构成法定人数，我们需要两个组及这两个组中各取两个服务器，也就是总共4个服务器。但根据法定人数多数原则，我们至少需要5个服务器来构成一个法定人数。注意，不能从任何子集形成法定人数的4个服务器，不过，一个组全部服务器加上另一个组的一个单独的服务器并不能构成法定人数。

当我们在跨多个数据中心部署ZooKeeper服务时，这种配置方式有很多优点。例如，一个组可能表示运行于不同数据中心的一组服务器，即使这个数据中心崩溃，ZooKeeper服务也可以继续提供服务。

在跨三个数据中心部署的这种方式可以容忍某个数据中心的故障问题，我们可以在两个数据中心中每一个部署三个服务器，而只在第三个数据中心部署一个服务器，通过这种方式来使用多数原则，这样，如果某个数据中心不可用，其他两个数据中心还能组成法定人数。这种配置方式的优点是这七个服务器中的任何四个都构成一个法定人数，而缺点是一旦某个数据中心不可用，其他数据中心中任何服务器的崩溃都无法容忍。

如果只有两个数据中心可用，我们可以使用权重值来表示优先权，例如，基于每个数据中心中的客户端数量来配置权重值。只有两个数据中心时，如果每个服务器的权重值都一样，我们就无法容忍任何一个数据中心的失效，但是如果我们对某个服务器分配了更高的权重值，我们就可以容忍这两个数据中心中某个数据中心的失效。假设，我们在每个数据中心中分配三个服务器，并且我们将这些服务器均放到同一组中：

```
group.1=1:2:3:4:5:6
```

因为所有的服务器默认情况下权重值都一样，因此只要6个服务器中有4个服务器有效时就可以构成法定人数的服务器。当然，这也意味着如果某个数据中心失效，我们就不能形成法定人数，即使另一个数据中心的三个服务器均有效。

为了给服务器分配不同的权重值，我们可以通过以下选型进行配置：

`weight.x=n`

与`group`选项一起配合使用，通过该选项可以为某个服务器形成法定人数时分配一个权重值为`n`。其值`n`为服务器投票时的权重，ZooKeeper中群首选举和原子广播协议中均需要投票。默认情况下，服务器的权重值为1，如果配置文件中定义了组的选项，但未指定权重值，所有的服务器均会被分配权重值1。

我们假设，某个数据中心只要其所有服务器均可用，即使在其他数据中心失效时，这个数据中心也可以提供服务，我们暂且称该数据中心为D1，此时，我们可以为D1中的某个服务器分配更高权重值，以便可以更容易与其他服务器组成法定人数。

假设D1中有服务器1、2和3，我们通过以下方式为服务器1分配更高的权重值：

```
weight.1=2
```

通过以上配置，我们就有了7个投票，在构成法定人数时，我们只需要4个投票。如果没有`weight.1=2`参数，任何服务器都需要与其他三个服务器来构成法定人数，但有了这个参数配置，服务器1与两个服务

器就可以构成法定人数。因此，只要D1可用，即使其他数据中心发生故障，服务器1、2和3也能构成法定人数并继续提供服务。

通过以上不同的法定人数配置的若干示例，我们看到该配置对部署的影响。我们提供的分层方案非常灵活，通过不同的权重值和组的管理可以提供不同的分层配置。

10.2.3 观察者

回忆之前说介绍的，观察者（observer）为ZooKeeper服务器中不参与投票但保障状态更新顺序的特殊服务器。配置ZooKeeper集群使用观察者，需要在观察者服务器的配置文件中添加以下行：

```
peerType=observer
```

同时，还需要在所有服务器的配置文件中添加该服务器的：
observer定义。如下：

```
server.1:localhost:2181:3181:observer
```

10.3 重配置

我们看到，配置涉及很多工作，不是吗？不过，现在你已经掌握了如何配置，你将要开始配置一个由三个不同的服务器组成的ZooKeeper集群，但是一两个月后，你发现使用ZooKeeper的客户端进程越来越多，并且成为一个更加关键的服务，因此你需要增加集群服务器到五个服务器，没什么大不了的，是嘛？你可以在深夜停止集群，重新配置所有服务器，然后不到一分钟之内恢复服务，如果应用程序恰当处理了Disconnected事件，用户可能不会感知到服务中断。我们在刚开始开发ZooKeeper时也是这么想的，但事实证明，情况要复杂得多。

考虑图10-1的场景，三个服务器（A、B、C）组成了整个集群，服务器C因某些网络拥塞问题稍稍落后于整个集群，因此服务器C刚刚了解事务到<1, 3>（其中1为时间戳，3为对应该时间戳的事务标识，在9.1节已经介绍过，但因为服务器A和B的通信良好，所以服务器C稍稍落后并不会导致整个系统变慢，服务器A和B可以提交事务到<1, 6>）。

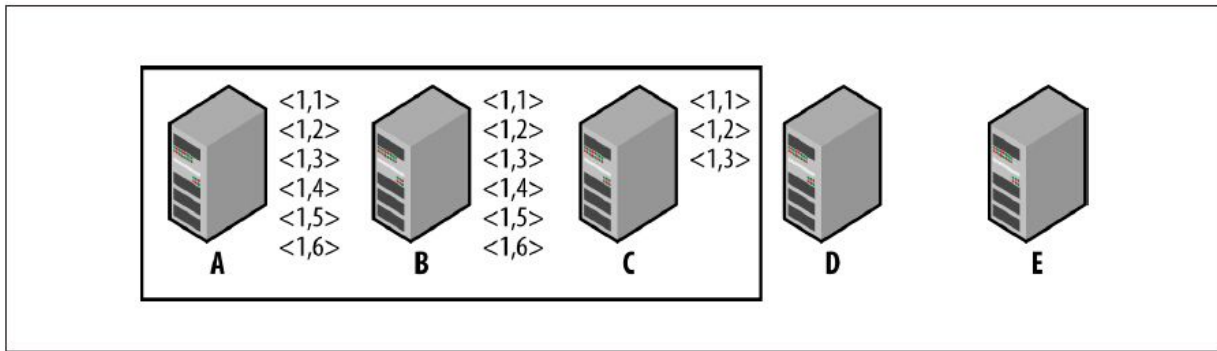


图10-1: 含有3个服务器的集群将要扩展到5个

现在，假设我们将所有服务停止，添加服务器D和E到集群中，当然这两个新的服务器并不存在任何状态信息，我们重新配置了服务器A、B、C、D、E成为更大的集群并启动集群恢复服务，因为我们现在有了五个服务器，我们至少需要三个服务器组成一个法定人数，而服务器C、D、E足够构成法定人数，因此在图10-2中我们看到当这些服务器构成法定人数并开始同步时都发生了什么。这个场景可以简单重现，如果服务器A和B的启动慢一些，比如服务器A和B比其他三个服务器的启动晚一些。一旦新的法定人数开始同步，服务器A和B就会与服务器C进行同步，因为法定人数中服务器C的状态为最新状态，法定人数的三个成员服务器会同步到最后的事务 $\langle 1, 3 \rangle$ ，而不会同步 $\langle 1, 4 \rangle$ 、 $\langle 1, 5 \rangle$ 和 $\langle 1, 6 \rangle$ 这三个事务，因为服务器A和B并未构成法定人数的成员。

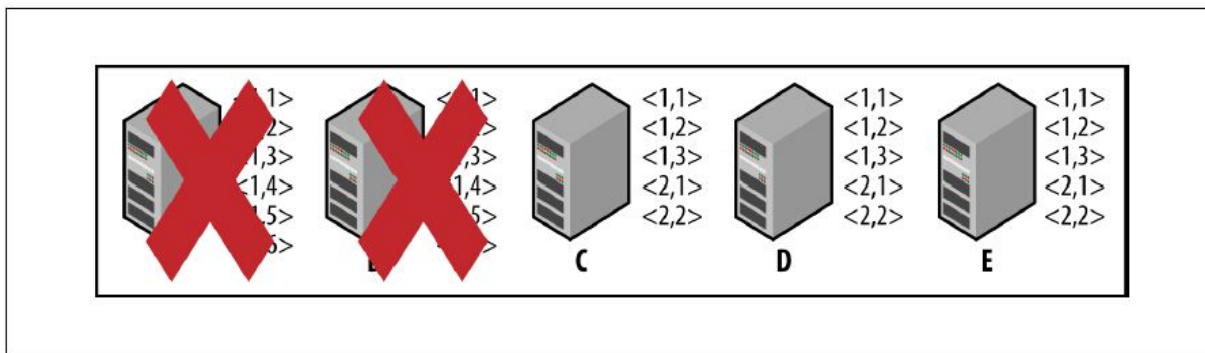


图10-2: 5个服务器的集群的法定人数为3

因为已经构成一个活跃的法定人数，这些服务器可以开始提交新的事务，我们假设有两个事务： $\langle 2, 1 \rangle$ 和 $\langle 2, 2 \rangle$ ，如图10-3所示，当服务器A和B启动后连接到服务器C后，服务器C作为群首欢迎其加入到集群之中，并在收到事务 $\langle 2, 1 \rangle$ 和 $\langle 2, 2 \rangle$ 后立即告知服务器A和B删除事务 $\langle 1, 4 \rangle$ 、 $\langle 1, 5 \rangle$ 和 $\langle 1, 6 \rangle$ 。

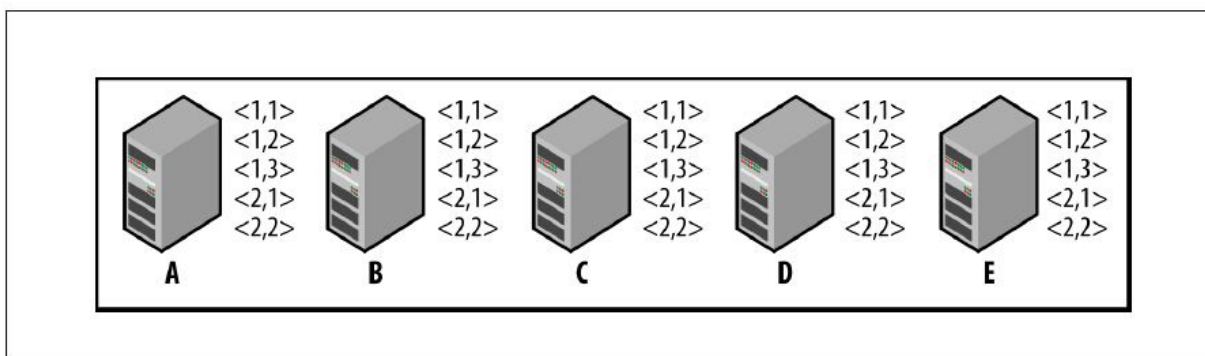


图10-3: 5个服务器的集群丢失数据

这个结果非常糟糕，我们丢失了某些状态信息，而且状态副本与客户端所看到的 $\langle 1, 4 \rangle$ 、 $\langle 1, 5 \rangle$ 、 $\langle 1, 6 \rangle$ 也不再一致。为了避免这个问题，ZooKeeper提供了重配置操作，这意味着运维人员并不需要手工

进行重配置操作而导致状态信息的破坏，而且，我们也不需要停止任何服务。

重配置不仅可以让我们改变集群成员配置，还可以修改网络参数配置，因为ZooKeeper中配置信息的变化，需要将重配置参数与静态的配置文件分离，单独保存为一个配置文件并自动更新该文件。

dynamicConfigFile参数和链接这两个配置文件。

注意：我是否可以使用动态配置选项？

该功能已经在Apache仓库的主干分支中被添加，主干的目标版本号为3.5.0，不过并不保证必然是这个版本号发布该功能，这取决于主干的进展情况，在本书完成时，最新的版本号为3.4.5，而该版本中尚未加入该功能。

我们使用动态配置之前，回顾一下之前的配置文件：

```
tickTime=2000
initLimit=10
syncLimit=5
dataDir=./data
dataLogDir=./txnlog
clientPort=2182
server.1=127.0.0.1:2222:2223
server.2=127.0.0.1:3333:3334
server.3=127.0.0.1:4444:4445
```

现在，我们将配置文件修改为动态配置方式：

```
tickTime=2000
initLimit=10
```

```
syncLimit=5
dataDir=./data
dataLogDir=./txnlog
dynamicConfigFile=./dyn.cfg
```

注意，我们甚至从配置文件中删除了`clientPort`参数配置，在`dyn.cfg`文件由服务器项的配置组成，同时还多了一些配置，服务器项的配置形式如下：

```
server.id=host:n:n[:role];[client_address:]client_port
```

与正常的配置文件一样，列出了每个服务器的主机名和端口号用于法定人数和群首选举消息。`role`选项必须为`participant`或`observer`，如果忽略`role`选项，默认为`participant`，我们还指定了`client_port`（用于客户端连接的服务器端口号），以及该服务器需要绑定的特定网络接口地址，因为我们从静态配置文件中删除了`clientPort`参数，所以我们在这里添加该配置。

因此，最终我们的`dyn.cfg`配置文件如下所示：

```
server.1=127.0.0.1:2222:2223:participant;2181
server.2=127.0.0.1:3333:3334:participant;2182
server.3=127.0.0.1:4444:4445:participant;2183
```

我们使用重配置之前必须先创建这些文件，一旦这些文件就绪，我们就可以通过`reconfig`操作来重新配置一个集群，该操作可以增量或全量（整体）地进行更新操作。

增量的重配置操作将会形成两个列表：待删除的服务器列表，待添加的服务器项的列表。待删除的服务器列表仅仅是一个逗号分隔服务器ID列表，待添加的服务器项列表为逗号分隔的服务器项列表，每个服务器项的形式为动态配置文件中所定义的形式。例如：

```
reconfig -remove 2,3 -add \  
server.4=127.0.0.1:5555:5556:participant;2184,\  
server.5=127.0.0.1:6666:6667:participant;2185
```

该命令将会删除服务器2和3，添加服务器4和5。该操作成功执行还需要满足某些条件，首先，与其他ZooKeeper操作一样，原配置中法定人数必须处于活动状态；其次，新的配置文件中构成的法定人数也必须处于活动状态。

注意：通过重配置从一个服务器到多个服务器

当我们只有一个单独的ZooKeeper服务器，该服务器以独立模式运行，这种情况稍微复杂一些，因为重配置不仅改变了法定人数组成的元素，同时还会切换原来的服务器模式从独立模式到仲裁模式，所以，我们不允许以独立模式运行重配置操作，只有在仲裁模式时才可以使用重配置功能。

ZooKeeper一次允许一个配置的变更操作请求，当然，配置操作会非常快地被处理，而且重新配置也很少发生，所以并发的重配置操作应该不是什么问题。

我们还可以使用 **-file** 参数来指定一个新的成员配置文件来进行一次全量更新。例如：**reconfig-file newconf** 命令会产生如上面命令一样的增量操作结果，**newconf** 文件为：

```
server.1=127.0.0.1:2222:2223:participant;2181
server.4=127.0.0.1:5555:5556:participant;2184
server.5=127.0.0.1:6666:6667:participant;2185
```

通过 **-members** 参数，后跟服务器项的列表信息，可以代替 **-file** 参数进行全量更新配置操作。

最后，所有形式的 **reconfig** 的为重新配置提供了条件，如果通过 **-v** 参数提供了配置版本号，**reconfig** 命令会在执行前确认配置文件当前的版本号是否匹配，只有匹配才会成功执行。你可以通过读取 **/zookeeper/config** 节点来获取当前配置的版本号，或通过 **zkCli** 工具来调用 **config** 获取配置版本号信息。

注意：手动重配置

如果你真想手动进行重配置操作（也许你使用旧版本的 ZooKeeper），最简单最安全的方式是，每次在停止整个服务器和启动 ZooKeeper 集群服务（即让群首建立起来）时只进行一个配置变更操作。

客户端连接串的管理

我们已经讨论了ZooKeeper服务器的配置问题，但对于客户端也涉及一些相关的配置问题：连接串。客户端连接串常常表示为逗号分隔的host: port对，其中host为主机名或IP地址，通过主机名可以提供服务器实际IP与所访问的服务器的标识符之间的间接层的对应关系。例如，运维人员可以替换ZooKeeper服务为另一个，而不需要改变客户端的配置。

不过，该灵活性有一定限制，运维人员可以改变组成集群的服务器机器，但不能改变客户端所使用的服务器。例如，如图10-4所示，ZooKeeper可以通过重配置很简单地将集群从三个服务器扩展到五个服务器，但客户端仍然使用三个服务器，而不是五个。

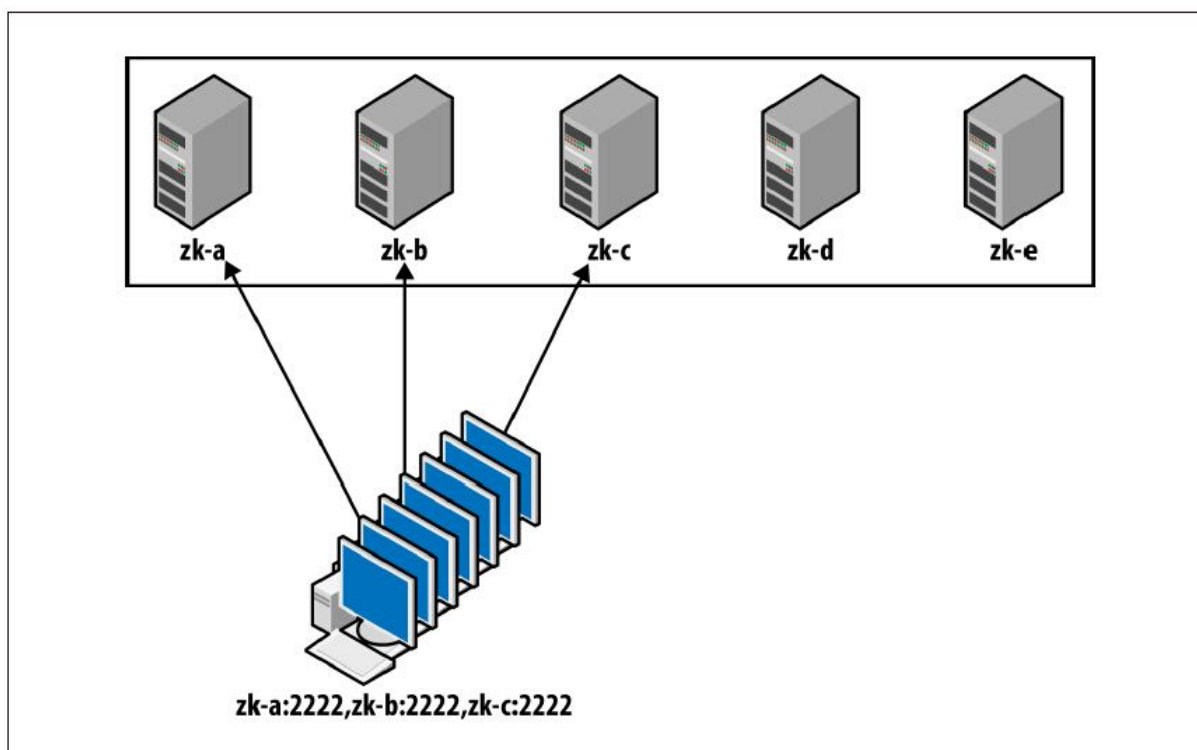


图10-4: 集群从三个到五个服务器时, 客户端的重配置

另一种方式可以使ZooKeeper的服务器数量更具弹性, 而不需要改变客户端的配置。对主机名我们很自然地想到可以解析为一个IP地址, 但实际上, 一个主机名可以解析为多个地址, 如果主机名解析为多个IP地址, ZooKeeper就可以连接到其中的任何地址, 在图10-4中, 假设服务器zk-a、zk-b和zk-c, 解析为三个独立的IP地址: 10.0.0.1、10.0.0.2和10.0.0.3, 现在假设你通过DNS配置了一个单独的主机名: zk, 解析为这三个IP地址, 你只需要修改DNS的解析地址数量, 之后启动的任何客户端都可以访问这五个服务器, 如图10-5所示。

在使用主机名解析为多个地址方式时, 还有一些注意事项。首先, 所有的服务器必须使用相同的客户端端口号; 其次, 主机名解析只有在创建连接时才会发生, 所以已经连接的客户端无法知道最新的名称解析, 只能对新创建的ZooKeeper客户端生效。

客户端的连接还可以包含路径信息, 该路径指示了解析路径名称时的根路径, 其行为与UNIX系统中的chroot命令相似, 而且在ZooKeeper社区中你也会经常听到人们以“chroot”来称呼这个功能。例如, 如果客户端的连接串为zk: 2222/app/superApp, 当客户端连接并执行getData ("/a.dat", ...) 操作时, 实际客户端会得到/app/superApp/a.dat节点的数据信息(注意, 连接串中指示的路径必须存在, 而不会为你创建连接串中所指示的路径)。

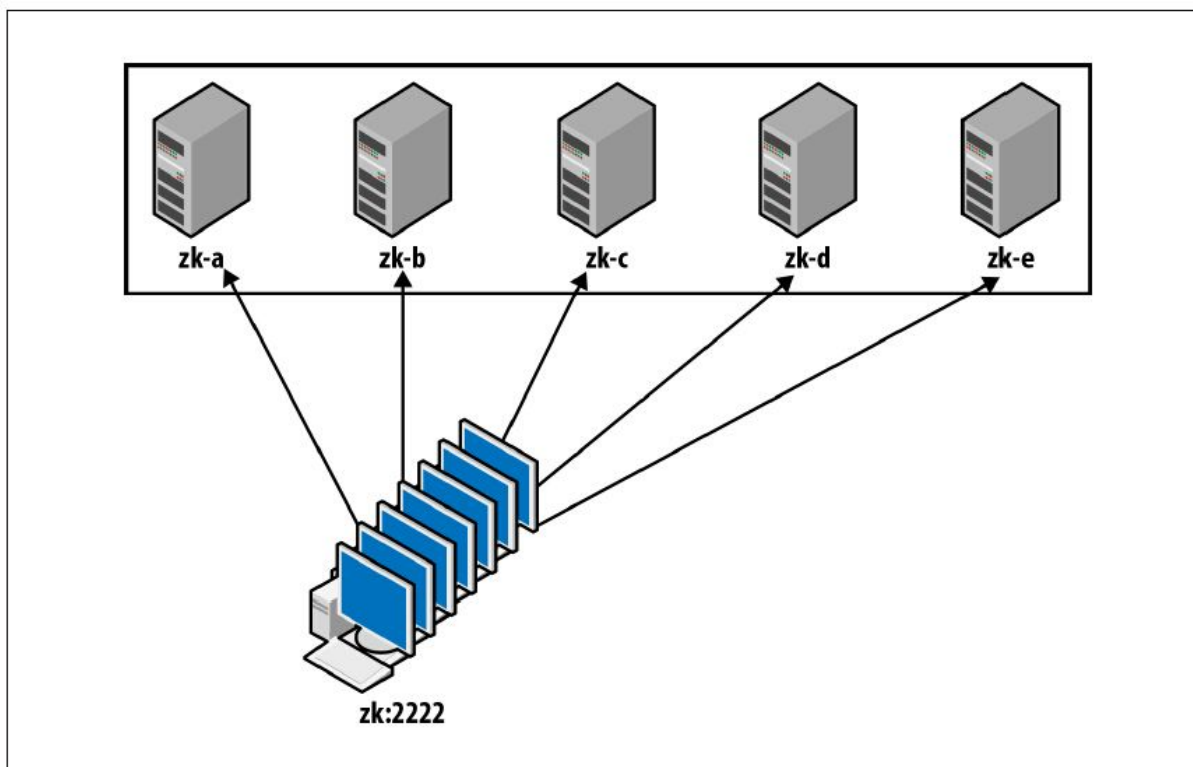


图10-5： 集群从三个到五个服务器时，使用DNS对客户端的重配置

在连接串中添加路径信息的动机在于一个ZooKeeper集群为多个应用程序提供服务，这样不需要要求每个应用程序添加其路径的前缀信息。每个应用程序可以类似名称独享似的使用ZooKeeper集群，运维人员可以按他们的期望来划分命名空间。图10-6的示例展示了不同的连接串可以为客户端应用程序提供不同的根入口点。

注意：连接串的重叠

当管理客户端连接串时，注意一个客户端的连接串永远不要包含两个不同的ZooKeeper集群的主机名，这是最快速也是最简单导致脑裂问题的方式。

10.4 配额管理

ZooKeeper的另一个可配置项为配额，ZooKeeper初步提供了znode节点数量和节点数据大小的配额管理的支持。我们可以通过配置来指定某个子树的配额，该子树就会被跟踪，如果该子树超过了配额限制，就会记录一条警告日志，但操作请求还是可以继续执行。此时，ZooKeeper会检测是否超过了某个配额限制，但不会阻止处理流程。

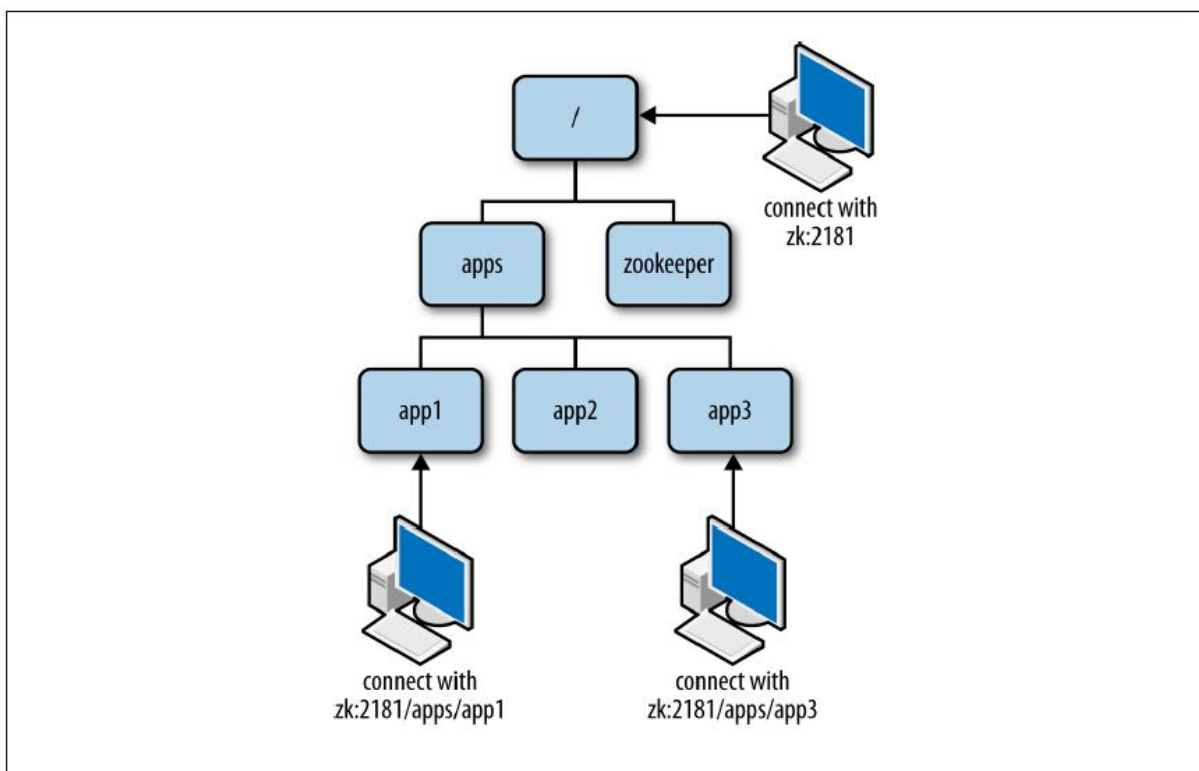


图10-6：通过连接串指定ZooKeeper客户端的根节点

配额管理的跟踪功能通过/zookeeper子树完成，所以应用程序不能在这个子树中存储自己的数据，这个子树只应该保留给ZooKeeper使用，而/zookeeper/quota节点就是ZooKeeper管理配额的节点。为了对应用程序/application/superApp创建一个配额项，我们需要在/application/superApp节点下创建两个子节点zookeeper_limits和zookeeper_stats。

对于znode节点数量的限制我们称之为count，而对于节点数据大小的限制则为bytes。在zookeeper_limits和zookeeper_stats节点中通过count=n，bytes=m来指定配额，其中n和m均为整数，在zookeeper_limits节点中，n和m表示将会触发警告的级别（如果配置为-1就不会触发警告信息），在zookeeper_stats节点中，n和m分别表示当前子树中的节点数量和子树节点的数据信息的当前大小。

注意：对元数据的配额跟踪

对于子树节点数据的字节数配额跟踪功能，并不会包含每个znode节点的元数据的开销，元数据的大小大约100字节，所以如果每个节点的数据大小都比较小，跟踪znode节点的数量比跟踪znode数据的大小更加实用。

我们可以使用zkCli来创建/application/superApp节点，并配置配额限制：

```
[zk: localhost:2181(CONNECTED) 2] create /application ""
Created /application
[zk: localhost:2181(CONNECTED) 3] create /application/superApp super
Created /application/superApp
[zk: localhost:2181(CONNECTED) 4] setquota -b 10 /application/superApp
Comment: the parts are option -b val 10 path /application/superApp
[zk: localhost:2181(CONNECTED) 5] listquota /application/superApp
absolute path is /zookeeper/quota/application/superApp/zookeeper_limits
Output quota for /application/superApp count=-1,bytes=10
Output stat for /application/superApp count=1,bytes=5
```

我们创建了/application/superApp节点，且该节点的数据为5个字节（一个单词“super”），之后我们为/application/superApp节点设置了配额限制为10个字节，当我们列出/application/superApp节点配置限制是，我们发现数据大小的配额还有5个字节的余量，而我们并未对这个子树设置znode节点数量的配额限制，因为配额中count的值为-1。

如果我们发送命令
get/zookeeper/quota/application/superApp/zookeeper_stats，我们可以直接访问该节点数据，而不需要使用zkCli工具，事实上，我们可以通过创建或删除这些节点来创建或删除配额配置。如果我们运行以下命令：

```
create /application/superApp/lotsOfData ThisIsALotOfData
```

我们就会在日志中看到如下信息：

```
Quota exceeded: /application/superApp bytes=21 limit=10
```

10.5 多租赁配置

配额，提供了配置选项中的某些限制措施，而ACL策略更值得我们考虑如何使用ZooKeeper来服务于多租赁（multitenancy）情况。满足多租赁的一些令人信服的原因如下：

- 为了提供可靠的服务器，ZooKeeper服务器需要运行于专用的硬件设备之上，跨多个应用程序共享这些硬件设备更容易符合资本投资的期望。

- 我们发现，在大多数情况下，ZooKeeper的流量非常具有突发性：配置或状态的变化的突发操作会导致大量的负载，从而导致服务长时间的不可用。如果是没有什么关联的应用程序的突发操作，将这些应用程序共享这个服务器更能有效利用硬件资源。不过还是要注意失联事件发生时所产生的峰值，某些写得不太规范的应用程序在处理Disconnected事件时，产生的负载高于其所需要的资源。

- 对于硬件资源的分摊，我们可以获得更好的故障容错性：如果两个应用程序，从之前各自三个服务器的集群中转移到一个由5台服务器组成的集群，总量上所使用的服务器更少了，对ZooKeeper也可以容忍两台服务器的故障，而不是之前的只能容忍一个服务器故障。

当服务于多租赁的情况下时，运维人员一般会将数据树分割为不同的子树，每个子树为某个应用程序所专用。开发人员在设计应用程序时可以考虑在其所用的`znode`节点前添加前缀，但还有一个更简单的方法来隔离各个应用程序：在连接串中指定路径部分，10.3节中介绍了这方式。每个应用程序的开发人员在开发应用程序时，就像使用专用的ZooKeeper服务一样。如果运维人员决定将应用程序部署到根路径`/application/newapp`之下，应用程序可以使用`host:`

`port/application/newapp`连接串，而不仅仅是`host: port`，通过这种方式，对应用程序所呈现的犹如使用专用服务一样，与此同时，运维人员还可以为`/application/newapp`节点配置配额限制，以便跟踪应用程序的空间使用情况。

10.6 文件系统布局和格式

我们已经讨论过快照、事务日志以及存储设备等问题，本节中，我们将会讨论这些信息在文件系统中如何配置。学习本节中所涉及的概念，需要对9.6节中所讨论的这些概念具有较好的理解，以便随时可以回顾相关的知识点。

我们之前已经介绍了，数据存储有两类：事务日志文件和快照文件。这些文件均以普通文件的形式保存到本地文件系统中，在进行关键路径的事务处理时就会写入事务日志文件，所以我们强烈建议将这些文件保存到一个专用存储设备上（事实上我们已经多次提到这个问题，因为这对于吞吐能力和延迟的一致性非常重要），不使用专用存储设备保存事物日志文件并不会导致任何正确性相关的问题，却会影响性能。在虚拟化环境中，也许无法获得专用存储设备。对于快照文件并不要求存储于专用存储设备上，因为该文件由后台线程慢慢写入。

快照文件将会被写入到**DataDir**参数所指定的目录中，而事务日志文件将会被写入到**DataLogDir**参数所指定的目录中。首先，我们看一下事务日志目录中的文件，如果你列出该目录的信息，你会发现只有一个子目录，名为**version-2**，我们对日志和快照的格式只做出了一次

重大改进，当我们改变其格式后，我们发现，将数据通过文件版本进行分离，对于处理版本间的数据迁移是非常有用的。

10.6.1 事务日志

让我们看一看在运行一些小测试后的目录内容，我们发现有两个事务日志文件：

```
-rw-r--r--  1 breed 67108880 Jun  5 22:12 log.1000000001  
-rw-r--r--  1 breed 67108880 Jul 15 21:37 log.2000000001
```

我们可以仔细观察这些文件信息。首先，考虑到测试很少，而这些文件却非常大（每个都超过6MB）；其次这些文件名的后缀中均有一个很大数字。

ZooKeeper为文件预分配大的数据块，来避免每次写入所带来的文件增长的元数据管理开销，如果你通过对这些文件进行十六进制转储打印，你会发现这些文件中全部以null字符（\0）填充，只有在最开始部分有少量的二进制数据，服务器运行一段时间后，其中的null字符逐渐被日志数据替换。

日志文件中包含事务标签zxid，但为了减轻恢复负载，而且为了快速查找，每个日志文件的后缀为该日志文件中第一个zxid的十六进制形式。通过十六进制表示zxid的一个好处就是你可以快速区分zxid中

时间戳部分和计数器部分，所以在之前例子中的第一个文件的时间戳为1，而第二个文件的时间戳为2。

不过，我们还想继续看一看文件中保存了什么内容，对于问题诊断也非常有帮助。有时，开发人员宣称ZooKeeper丢失了某些znode节点信息，此时只有通过查找事务日志文件才可以知道客户端具体删除过哪些节点。

我们可以通过一下命令来查看第二个日志文件：

```
java -cp $ZK_LIBS org.apache.zookeeper.server.LogFormatter version-2 /  
log.2000000001
```

这个命令的输出信息如下：

```
7/15/13... session 0x13...00 cxid 0x0 zxid 0x2000000001 createSession 30000  
7/15/13... session 0x13...00 cxid 0x2 zxid 0x2000000002 create  
'/test,#22746573746 ...  
7/15/13... session 0x13...00 cxid 0x3 zxid 0x2000000003 create  
'/test/c1,#6368696c ...  
7/15/13... session 0x13...00 cxid 0x4 zxid 0x2000000004 create  
'/test/c2,#6368696c ...  
7/15/13... session 0x13...00 cxid 0x5 zxid 0x2000000005 create  
'/test/c3,#6368696c ...  
7/15/13... session 0x13...00 cxid 0x0 zxid 0x2000000006 closeSession null
```

每个日志文件中的事务均以可读形式一行行地展示出来。因为只有变更操作才会被记录到事务日志，所以在事务日志中不会看到任何读事务操作。

10.6.2 快照

快照文件的命名规则与事务日志文件的命名规则相似，以下为之前例子中的服务器的快照列表信息：

```
-rw-r--r--  1 br33d  296 Jun  5 07:49 snapshot.0
-rw-r--r--  1 br33d  415 Jul 15 21:33 snapshot.1000000009
```

快照文件并不会被预分配空间，所以文件大小也更加准确地反映了其中包含的数据大小。其中后缀表示快照开始时当时的zxid值，我们之前已经介绍过，快照文件实际上为一个模糊快照，直到事务日志重现之后才会成为一个有效的快照文件。因此在恢复系统时，你必须从快照后缀的zxid开始重现事务日志文件，甚至更早的zxid开始重现事务。

快照文件中保存的模糊快照信息同样为二进制格式，因此，我们可以通过另一个工具类来检查快照文件的内容：

```
java -cp ZK_LIBS org.apache.zookeeper.server.SnapshotFormatter version-2 /
snapshot.1000000009
```

这个命令的输出信息如下：

```
----
/
cZxid = 0x0000000000000000
ctime = Wed Dec 31 16:00:00 PST 1969
mZxid = 0x0000000000000000
mtime = Wed Dec 31 16:00:00 PST 1969
```

```

pZxid = 0x00000100000002
cversion = 1
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0000000000000000
dataLength = 0
----
/sasd
cZxid = 0x00000100000002
ctime = Wed Jun 05 07:50:56 PDT 2013
mZxid = 0x00000100000002
mtime = Wed Jun 05 07:50:56 PDT 2013
pZxid = 0x00000100000002
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0000000000000000
dataLength = 3
----
....

```

只有每个节点的元数据被转储打印出来，这样，运维人员就可以知道一个znode节点何时发生了变化，以及哪个znode节点占用了大量内存。很遗憾，数据信息和ACL策略并没有出现在输出中，因此，在进行问题诊断时，记住将快照中的信息与日志文件的信息结合起来分析问题所在。

10.6.3 时间戳文件

ZooKeeper的持久状态由两个小文件构成，它们是两个时间戳文件，其文件名为acceptedEpoch和currentEpoch，我们之前已经讨论过时间戳的概念，而这两个文件则反映了某个服务器进程已接受的和正在处理的信息。虽然这两个文件并不包含任何应用数据信息，但对于数据一致性却至关重要，所以如果你在备份一个ZooKeeper服务器的原始数据文件时，不要忘了这两个文件。

10.6.4 已保存的ZooKeeper数据的应用

ZooKeeper数据存储的一个优点是，不管独立模式的服务器还是集群方式的服务器，数据的存储方式都一样。我们之前已经介绍过通过事务日志和快照的合并可以得到准确的数据视图，你可以将事务日志文件和快照文件拷贝到另一台设备上这些操作（例如在你的便携电脑中），将这些文件放到一个独立模式的服务器下空白的data目录下，然后启动服务，该服务就会真实反映出你所拷贝的那个服务器上的状态信息。这项技术可以使你从生产环境拷贝服务器的状态信息，用于稍后的复查等用途。

同时也意味着，你只需要简单地将这些数据文件进行备份，就可以轻易地完成ZooKeeper服务器的备份，如果你采用这种方式进行备份，还需要注意一些问题。首先，ZooKeeper为复制服务，所以系统中存在冗余信息，如果你进行备份操作，你只需要备份其中一台服务器的数据信息。

当ZooKeeper服务器认可了一个事务，从这时起它就会承诺记录下该状态信息，你一定要记住这一点，这一点非常重要。因此如果你使用旧的备份文件恢复一个服务器，就会导致服务器违反其承诺。如果你刚刚遭遇了所有服务器的数据丢失的情况，这可能不是什么大问

题，但如果你的集群在正常工作中，而你为某个服务器还原为旧的状态，你的行为可能会导致其他服务器也丢失了某些信息。

如果你要对全部或大多数服务器进行数据丢失的恢复操作，最好的办法是使用你最新抓取的状态信息（从最新的存活服务器中获取的备份文件），并在启动服务器之前将状态信息拷贝到其他所有服务器上。

10.7 四字母命令

现在，我们已经配置好服务器并启动运行，我们现在需要监控这些服务器，因此需要用到一系列四字母命令。在3.2.2节中曾经涉及了一些四字母命令的例子，我们通过telnet工具连接后，使用这些命令查看系统状态。四字母命令提供的这一简单方法，可以让我们对系统进行各种各样的检查。四字母命令的主要目标就是提供一个非常简单的协议，使我们使用简单的工具，如telnet或nc，就可以完成系统健康状况检查 and 问题的诊断。为简单起见，四字母命令的输出也是可读形式，使得更容易使用这些命令。

对服务器添加一个新的命令也很容易，命令列表也就会增长。本节中将会介绍一些常用的命令，对于最新的全部命令列表信息，请参考ZooKeeper文档。

ruok

提供（有限的）服务器的状态信息。如果服务器正在运行，就会返回imok响应信息。事实上“OK”状态只是一个相对的概念，例如，服务器运行中，虽无法与集群中其他服务器进行通信，然而该服务器返回的状态仍然是“OK”。对于更详细信息及可靠的健康状态检查，需要使用stat命令。

stat

提供了服务器的状态信息和当前活动的连接情况，状态信息包括一些基本的统计信息，还包括该服务器当前是否处于活动状态，即作为群首或追随者，该服务器所知的最后的zxid信息。某些统计信息为累计值，我们可以使用srst命令进行重置。

srvr

提供的信息与stat一样，只是忽略了连接情况的信息。

dump

提供会话信息，列出当前活动的会话信息以及这些会话的过期时间。该命令只能在群首服务器上运行。

conf

列出该服务器启动运行所使用的基本配置参数。

envi

列出各种各样的Java环境参数。

mntr

提供了比stat命令更加详细的服务器统计数据。每行输出的格式为key<tab>value。（群首服务器还将列出只用于群首的额外参数信息）。

wchs

列出该服务器所跟踪的监视点的简短摘要信息。

wchc

列出该服务器所跟踪的监视点的详细信息，根据会话进行分组。

wchp

列出该服务器所跟踪的监视点的详细信息，根据被设置监视点的znode节点路径进行分组。

cons, crst

cons命令列出该服务器上每个连接的详细统计信息，crst重置这些连接信息中的计数器为0

10.8 通过JMX进行监控

四字母命令可以用于系统的监控，但却没有提供系统控制和修改的方法，ZooKeeper通过标准Java管理协议，JMX（Java管理扩展），提供了更强大的监控和管理功能。有很多书籍介绍过如何配置和使用JMX，也有很多工具可用于JMX服务器管理，本节中，我们将会使用一个简单的管理控制台工具jconsole来探索通过JMX管理ZooKeeper功能。

jconsole为Java中自带的工具，实际上，类似jconsole这样的JMX工具常常用于监控远程的ZooKeeper服务器，但出于说明的目的，我们将会在ZooKeeper服务所运行的设备运行该工具。

首先，我们启动第二个ZooKeeper服务器（即ID为2的服务器），之后，我们只需要在命令行中简单的输入jconsole命令就可以启动jconsole工具，jconsole启动后，你就会看到类似图10-7中所示的窗口。

注意到其中带有“zookeeper”名称的进程，对于本地进程，jconsole会自动发现可连接的进程。

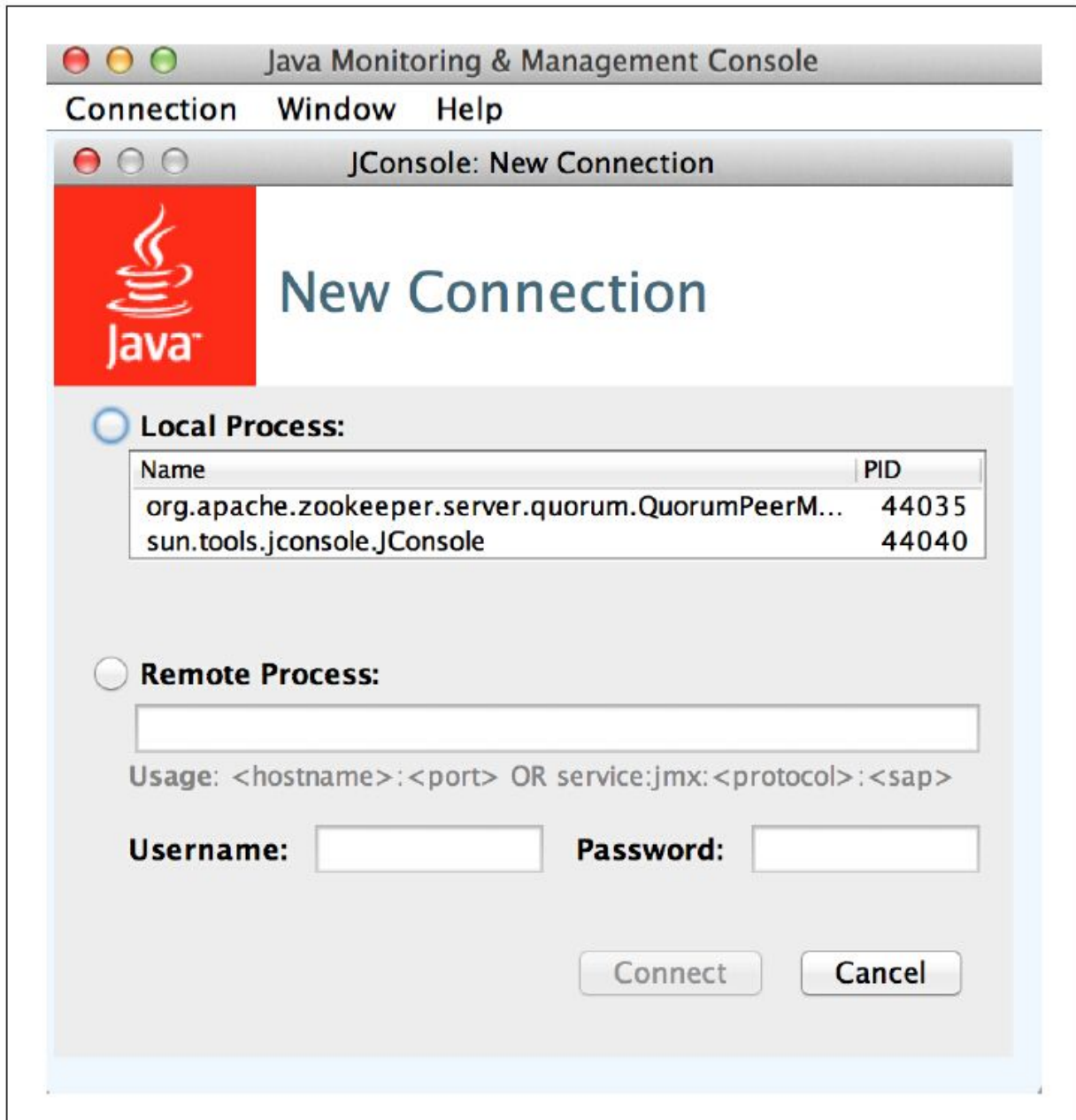


图10-7: jconsole启动界面

现在，我们只需要在列表中双击该进程就可以连接到ZooKeeper进程上，因为我们没有启用SSL，此时会提示我们关于非安全连接的选项，单击非安全连接按钮，之后屏幕中就会出现图10-8所示的窗口。

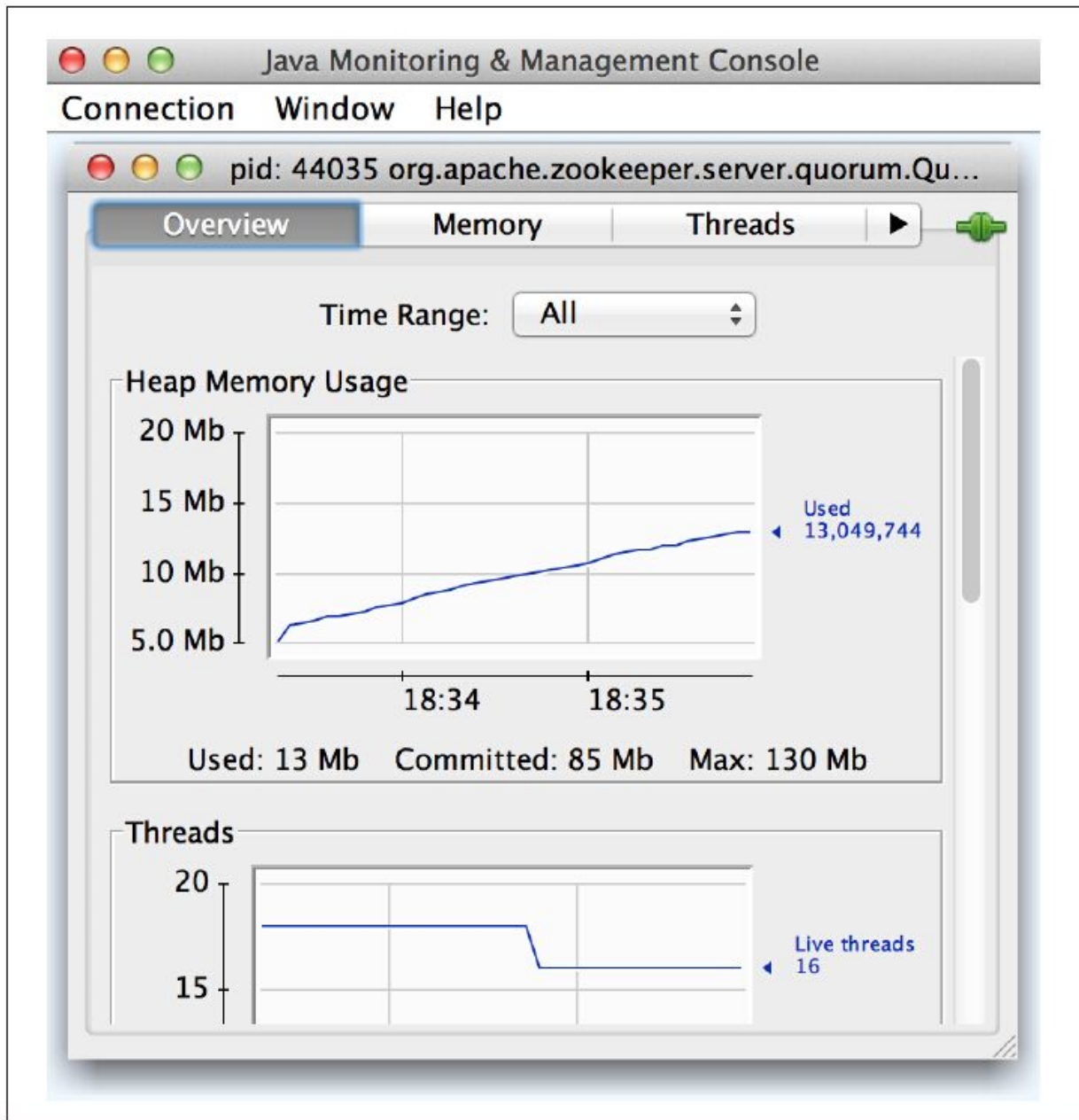


图10-8: 进程管理的第一个窗口

从这个界面中我们看到，我们可以通过该工具获取关于ZooKeeper服务器的各种各样有趣的统计信息。JMX支持通过MBean（托管Bean）来将自定义信息暴露给远程管理者，虽然名字听起来比较笨

拙，但却是暴露信息和操作的一个非常灵活的方式。jconsole会在最右侧的信息标签中列出进程暴露的所有MBean信息，如图10-9所示。

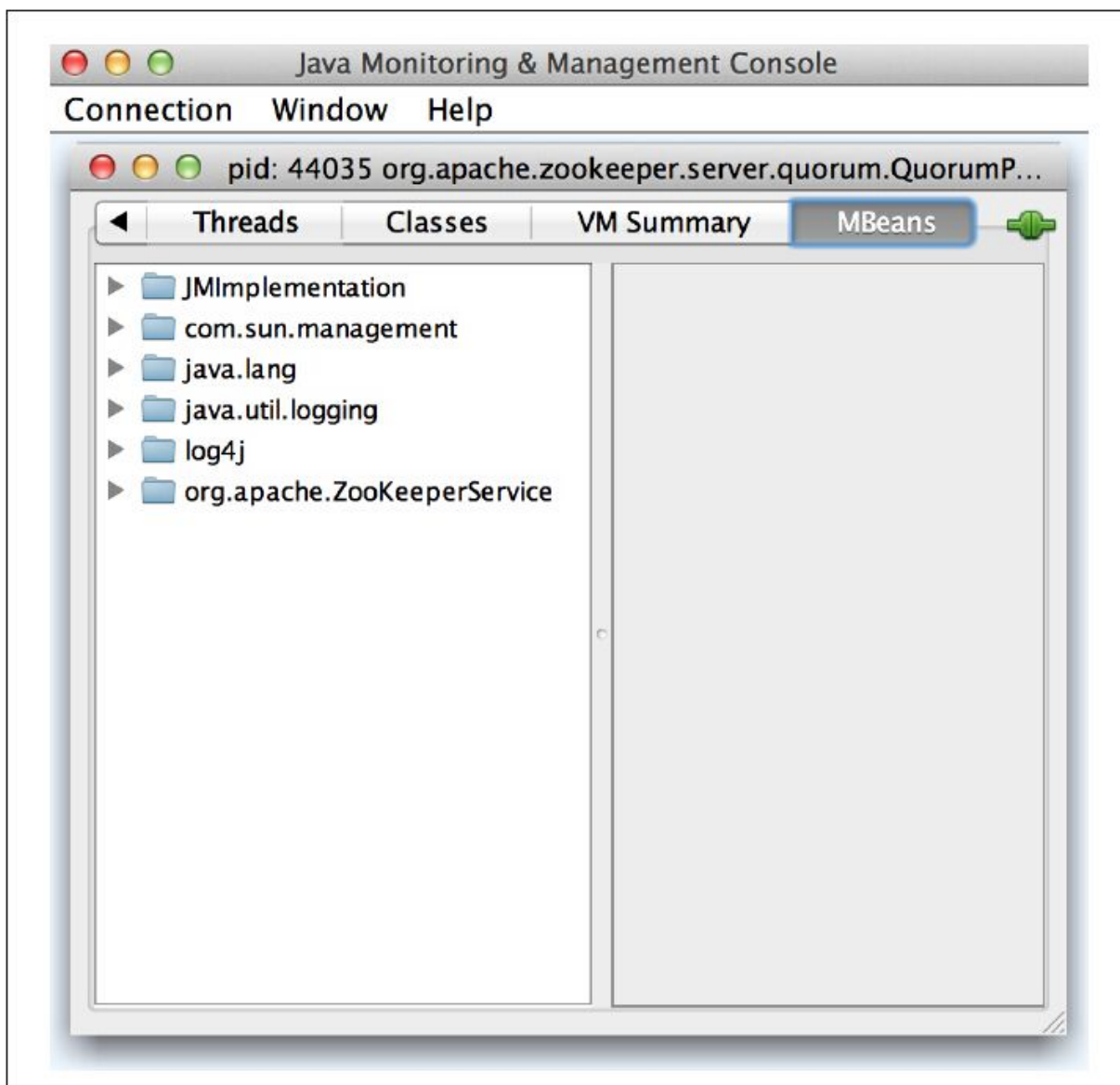


图10-9: jconsole中MBean信息

在MBean列表中我们可以看到ZooKeeper所使用且暴露出来的组件信息，我们比较关心ZooKeeperService的信息，因此我们双击该列表

项，我们将会看到一个分级的列表副本以及这些副本的信息，如果我们打开某些列表中的子项，我们会看到图10-10所示的信息。

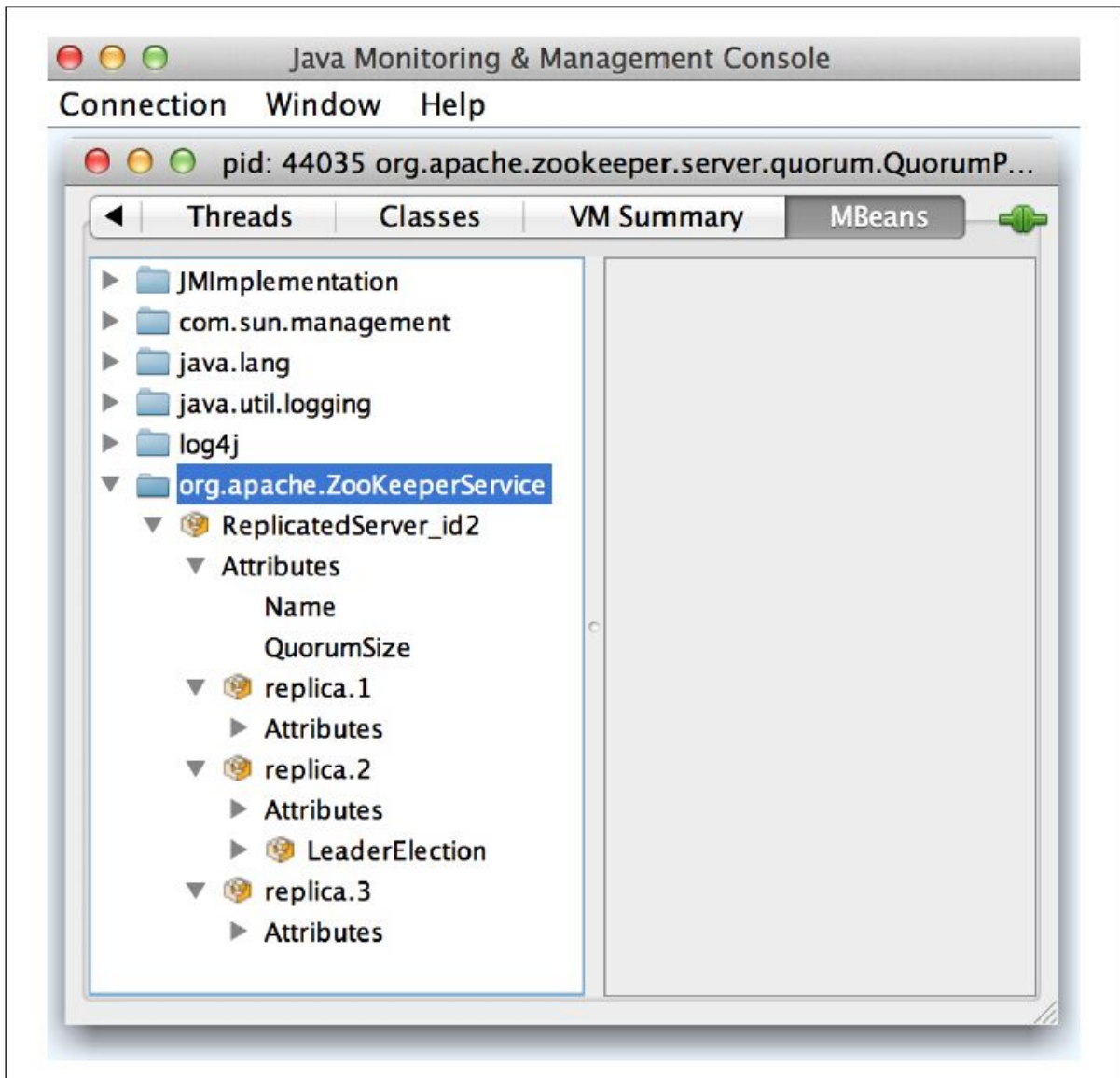


图10-10: jconsole中关于服务器2的信息

通过浏览replica.2的信息，我们注意到这些信息中还包括其他副本的信息，但只是一些通信信息，因为服务器2对其他副本所知信息并不

多，所以服务器2无法展示更多其他副本信息，而服务器2很了解自己的信息，所以它可以暴露更多的信息。

当我们启动服务器1，服务器2就可以与服务器1构成一个法定人数，此时我们就可以看到服务器2的更多信息。启动服务器1，之后再次通过jconsole检查服务器2信息。图10-11展示了通过JMX暴露的一些额外信息，我们可以看到服务器2当前角色为追随者，我们还可以看到数据数的信息。

图10-11还展示了服务器1的一些信息，我们看到，服务器1的角色为群首角色，同时还有一些额外信息，仅在群首服务器中，FollowerInfo中还会展示追随者的列表。当我们点击该按钮，我们会看到连接到服务器1的其他ZooKeeper服务器的原始列表信息。

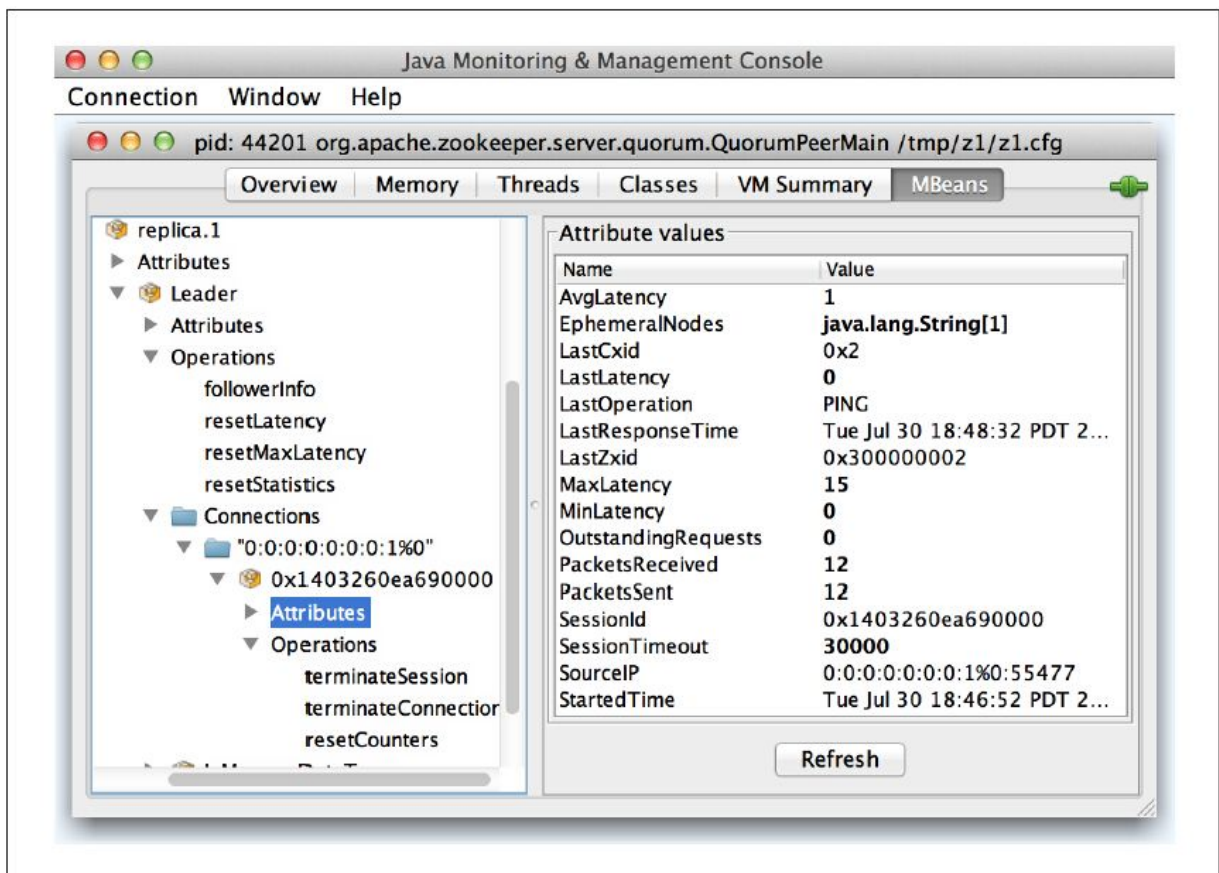


图10-11: jconsole中关于服务器1的信息

到目前为止，我们看到相比四字母命令，通过JMX所看到的信息更加优美直观，但是我们还没有看到有什么新功能，现在让我们看看JMX可以做到而四字母命令无法做到的功能。启动zkCli脚本工具，连接到服务器1，之后我们运行以下命令：

```
create -e /me "foo"
```

通过该命令，我们会创建一个临时性的znode节点，图10-11所示的服务器1的JMX信息中，我们可以看到出现了一个关于连接的新信息

项，连接的属性中列出了各种各样的信息，这些信息对于我们调试运行问题非常有用。这个视图中，我们还看到两个有意思的操作：

`terminateSession`和`terminateConnection`。

`terminateConnection`操作会关闭ZooKeeper客户端到服务器之间的连接，而会话依然处于活跃状态，所以此时客户端还可以重新连接到另一个服务器，客户端会收到失去连接的事件，但可以轻易恢复。

与之相反，`terminateSession`操作会声明会话已经死亡，客户端与服务器之间的连接将会被关闭，且会话也将因过期而中止，客户端也不能再使用这个会话与其他服务器建立连接。因此，使用`terminateSession`操作时需要小心，因为该操作会在会话超时之前就导致会话过期，所以在该进程自己发现过期前，其他进程可能已经发现了该进程的会话死亡的情况。

远程连接

JMX代理器运行于ZooKeeper服务器的JVM之中，如果连接远程的ZooKeeper服务器，我们需要配置好JMX代理器。对与远程连接的JMX协议有若干参数需要配置，本节中，我们展示了一种JMX的配置方式，来看看JMX提供了什么样功能。如果你在生产环境使用JMX，你可能需要使用另一种JMX配置——具体参考如何配置更高级的安全功能。

对JMX的配置可以通过系统属性的方式进行配置，在我们用于启动ZooKeeper服务器的zkServer.sh脚本中，我们使用SERVER_JVMFLAGS环境变量来配置这些系统属性。

例如，我们以下面的配置启动服务，我们就可以远程连接服务器3的55555端口。

```
SERVER_JVMFLAGS="-Dcom.sun.management.jmxremote.password.file=passwd \  
-Dcom.sun.management.jmxremote.port=55555 \  
-Dcom.sun.management.jmxremote.ssl=false \  
-Dcom.sun.management.jmxremote.access.file=access" \  
_path_to_zookeeper_/bin/zkServer.sh start _path_to_server3.cfg_
```

系统属性参数中用到了密码文件和访问控制文件，这些文件的格式非常简单。首先，创建passwd文件，方式如下：

```
# user password  
admin <password>
```

注意，密码文件中的密码信息为明文保存，因此只能给密码文件的所有者分配读写权限，如果不这样做，Java就无法启动服务。同时，我们关闭了SSL功能，这意味着密码信息在网络上会以明文的方式进行传输，如果我们需要更强的安全级别，JMX也提供了更强有力的选项，但这些已经超出本书所涉及的范围。

对于访问控制文件，我们将会给admin用户赋予readwrite权限，创建该文件的方式如下：

admin readwrite

10.9 工具

在ZooKeeper的发行包中提供了很多工具和软件，而有些工具或软件则单独发布，我们之前已经提到过日志格式化软件和Java中的JMX工具。在ZooKeeper发行包的contrib目录中，你可以找到一些软件，这些软件可以帮你集成ZooKeeper到其他的检测系统中去。我们列出了一部分最受欢迎的软件或工具：

- 通过C绑定实现的Perl和Python语言的绑定库。
- ZooKeeper日志可视化的软件。
- 一个基于网页的集群节点浏览和ZooKeeper数据修改功能的软件。
- ZooKeeper中自带的zktreeutil和guano均可以从GitHub下载。这些软件可以对ZooKeeper的数据进行导入和导出操作。
- zktop，也可以从GitHub下载，该软件监控ZooKeeper的负载，并提供Unix的top命令类似的输出。
- ZooKeeper冒烟测试，可以从GitHub上下载。该软件对ZooKeeper集群提供了一个简单的冒烟测试客户端，这个工具对于开发人员熟悉

ZooKeeper非常不错。

当然，这些并不是所有工具和软件的详细清单，还有很多已经开发的强大工具和软件不在ZooKeeper的发行包之中。如果你是ZooKeeper运维人员，可以在你的环境中多尝试各种各样的工具，这样做对你非常有意义。

10.10 小结

虽然ZooKeeper非常容易上手，但是在你的环境中还是有很多参数需要微调，ZooKeeper的可靠性和性能取决于正确的配置，因此了解ZooKeeper是如何工作的以及各个参数的具体含义非常重要。如果参数中的时间、法定人数（quorum）配置合理，ZooKeeper可以适应很多种网络拓扑结构。虽然手工修改ZooKeeper的成员信息非常危险，但通过ZooKeeper中提供reconfig操作就非常简单。现在有很多工具可以简化你的工作，所以花点时间多挖掘一些我们没有介绍到的工具。

作者介绍

Flavio Junqueira 是微软研究院在英国剑桥大学的研究人员之一。他拥有美国加州大学圣地亚哥分校计算机科学博士学位。他的研究范围涉及分布式系统的各个方面，包括分布式算法、并发性和可扩展性。他是Apache项目如Apache ZooKeeper（PMC主席和提交者）和Apache BookKeeper（提交者）的积极贡献者。他一有空就喜欢睡觉。

Benjamin Reed 是一位负责Facebook中所有细节工作的软件工程师。他以前的职位包括雅虎研究院首席研究科学家（负责所有大的方向）和IBM Almaden Research的研究人员（负责所有事情，无论大小）。他拥有加州大学圣克鲁斯分校计算机科学博士学位。他从事的工作涉及分布式计算、大数据处理、分布式存储、系统管理和嵌入式框架等领域。他参加了各种开源项目，如Hadoop和Linux操作系统等。他帮助启动了由Apache软件基金会主办的项目如Pig、ZooKeeper和BookKeeper。

封面介绍

本书封面上的动物是欧洲野猫（斑猫，学名：**Felis silvestris**），这是一种栖息在欧洲森林和草原以及土耳其和高加索山脉的野猫亚种。

体型与大家猫相似，欧洲野猫宽头、长毛和短尾——在喉咙、胸部和腹部有白斑。欧洲野猫的主食包括小型啮齿类动物如木鼠、田鼠、水鼠和树鼯。有趣的是，与爱吃鱼的家养猫的区别是，欧洲野猫鱼在野外很少捕鱼吃。

欧洲野猫曾一度遍布欧洲，被认为是最古老的种类之一——有限的化石记录表明欧洲野猫的历史可以追溯到更早的更新世时期。在过去300年中，由于受到狩猎以及人口扩张的影响，欧洲野猫的数量已经大幅减少。

杂交也是数量减少的一个大因素。虽然许多野猫亚种生活在偏远地区，但是有些则在相对靠近人类居住地，因此它们经常与周围的家养猫交配。这样很长一段时间后，某些亚种很可能会消失。